

Arca2 CPU Core

Reference Manual

Revision: 1.0

Jan. 2003



方舟科技有限公司

ARCA Technology Corporation

Arca2

CPU Core Reference Manual

Copyright © ARCA Technology Corporation. 2003.

Third party brands, logos and names are the property of those respective third parties.

Release history

Date	Revision	Change
Feb 2003	V1.0	

Disclaimer

This documentation is provided for use with ARCA Technology Corporation products. No license to ARCA Technology Corporation property rights is granted. ARCA Technology Corporation assumes no liability, provides no warranty either expressed or implied relating to the usage, or intellectual property right infringement except as provided for by the ARCA Technology Corporation Terms and Conditions of Sale.

ARCA Technology Corporation products are not designed for and should not be used in any medical or life sustaining or supporting equipment.

All information in this document should be treated as preliminary. ARCA Technology Corporation may make changes to this document without notice. Anyone relying on this documentation should contact ARCA Technology Corporation for the current documentation and errata.

ARCA Technology Corporation

5th Floor, Jade Palace Building,
76 Zhichun Rd, Haidian,
Beijing, P. R. China
Tel: 86-10-62638192
Fax:86-10-62638348
Http: www.arca.com.cn

Table of Contents

1	OVERVIEW	1
1.1	INTRODUCTION.....	1
1.2	BLOCK DIAGRAM.....	2
1.3	FEATURES.....	3
2	ARCA2 CPU.....	4
2.1	OVERVIEW.....	4
2.2	PIPELINE FOR ONE-PASS AND MULTI-PASS INSTRUCTIONS	5
2.2.1	<i>MULU with high 32-bit result.....</i>	5
2.2.2	<i>ITLB/ICACHE.....</i>	5
2.2.3	<i>SLEEP</i>	5
2.3	HAZARD & FORWARDING.....	6
2.3.1	<i>Forwarding method.....</i>	6
2.3.2	<i>Hazard.....</i>	6
2.4	CYCLES FOR ARCA2 INSTRUCTION EXECUTIONS AND STALLS.....	9
3	EXCEPTION MODEL	10
3.1	OVERVIEW.....	10
3.2	EXCEPTION TYPES.....	10
3.3	EXCEPTION PRIORITIES	12
3.4	EXCEPTION VECTOR TABLE	13
3.5	EXCEPTION CAUSE.....	14
3.6	CONTROL REGISTER	15
3.6.1	<i>Status Register (SR)</i>	15
3.6.2	<i>Spot-saving Register.....</i>	15
3.7	EXCEPTION ACKNOWLEDGEMENT PROCESS	16
3.8	RETURN FROM EXCEPTION ROUTINE	16
4	CORE CONFIGURATION	17
4.1	OVERVIEW.....	17
4.2	CONFIGURATION INSTRUCTIONS.....	18
4.2.1	<i>CLD/CST instruction.....</i>	18
4.2.2	<i>ITLB/DTLB/ICACHE/DCACHE instructions.....</i>	19
4.2.3	<i>CMD in ITLB/DTLB/ICACHE/DCACHE.....</i>	19
5	MEMORY MANAGE UNIT	21
5.1	OVERVIEW.....	21
5.1.1	<i>Features.....</i>	21
5.2	REGISTER CONFIGURATION.....	22
5.2.1	<i>Register Descriptions.....</i>	23
5.3	MEMORY SPACE.....	26
5.3.1	<i>Direct Map Virtual Address Space.....</i>	26
5.3.2	<i>Virtual Address Space In Paging System.....</i>	28
5.4	CONFIGURATION OF THE TLB	30
5.5	ADDRESS TRANSLATION METHOD	32
5.6	CONFIGURE OPERATION.....	34
5.6.1	<i>MMU Function.....</i>	34
5.6.2	<i>MMU Interface Format.....</i>	34
5.6.3	<i>Code Examples.....</i>	35
5.7	MMU EXCEPTION.....	37
5.7.1	<i>Illegal configure exception.....</i>	37
5.7.2	<i>Address Error</i>	37
5.7.3	<i>TLB Miss.....</i>	38
5.7.4	<i>Initial Page Write.....</i>	38

6	CACHE.....	39
6.1	OVERVIEW.....	39
6.1.1	Cache Feature.....	39
6.2	REGISTER CONFIGURATION.....	40
6.3	DATA CACHE AND WRITE BUFFER.....	41
6.3.1	D-cache Structure.....	42
6.3.2	Cacheable Access Operation.....	42
6.3.3	Non-cacheable Access Operation.....	44
6.3.4	Write Buffer.....	44
6.4	INSTRUCTION CACHE.....	45
6.4.1	Fetch Operation.....	45
6.5	PREFETCH OPERATION.....	46
6.6	SWAP OPERATION.....	47
6.7	ALIAS SOLUTION.....	48
6.8	COHERENCY BETWEEN CACHE AND EXTERNAL MEMORY.....	49
6.9	CACHE REPLACEMENT AND LOCK FUNCTION.....	50
6.10	CACHE CONFIGURATION.....	51
6.10.1	Operation List.....	51
6.10.2	Code Examples.....	53
7	DEBUG AND JTAG	56
7.1	OVERVIEW.....	56
7.1.1	Debug Features.....	57
7.1.2	Extended JTAG Feature.....	57
7.1.3	Debugging Pattern.....	57
7.1.4	Debug & JTAG Solution Diagram.....	58
7.2	EXTENDED JTAG.....	59
7.2.1	Overview.....	59
7.2.2	Standard & Extended Private Instructions.....	60
7.2.3	Extended Data Registers.....	61
7.2.4	Endian Adjustment.....	62
7.2.5	JTAG Memory Space.....	62
7.2.6	Miscellaneous Constraints	62
7.3	DEBUG MODULE.....	64
7.4	DEBUG REGISTER CONFIGURATION	65
7.4.1	Register Descriptions.....	65
7.5	DEBUG OPERATION.....	70
7.5.1	Overview.....	70
7.5.2	Fetch Breakpoint Operation.....	71
7.5.3	Data Access Breakpoint Operation.....	71
7.5.4	Asynchronous Break/Boot Operation.....	73
7.6	DEBUG EXCEPTION OPERATION.....	74
7.6.1	Fetch Breakpoint Debug Exception.....	75
7.6.2	Data Access Breakpoint Debug Exception.....	75
7.6.3	Software Breakpoint Debug Exception.....	76
7.6.4	Asynchronous Break Debug Exception	76
7.6.5	Asynchronous Boot Debug Exception.....	77
7.7	EXAMPLE FOR APPLICATION.....	78
7.7.1	Single step execution.....	78
7.7.2	Combinatorial Break Condition Capture	79
7.7.3	Data transfer between target and host	80
7.7.4	How To Access JTAG Memory Space	81
7.7.5	How To Implement Burst Access from JTAG memory (burst read 8 words / burst write 4 words).....	87
7.7.6	How To Boot System From JTAG Memory	87
8	PERFORMANCE MONITOR.....	89

8.1	OVERVIEW.....	89
8.2	REGISTER CONFIGURATION.....	90
8.2.1	<i>Performance Monitor Control Register (PMC)</i>	90
8.2.2	<i>Clock Cycle Time Register (CTR)</i>	91
8.2.3	<i>Monitor Object Counter Register 0 (MOR0)</i>	91
8.2.4	<i>Monitor Object Counter Register 1 (MOR1)</i>	92
8.3	MONITORING EVENT.....	93
8.4	MONITORING FLOW.....	95
LIST OF FIGURES		96
LIST OF TABLES.....		97

1 Overview

1.1 Introduction

Arca2 CPU core is a high performance and low power microprocessor core which implements version 2 of *Arca Instruction Set Architecture*. Refer the document “*Arca Instruction Set Architecture Reference Manual-V2*” for ISA details.

The CPU core is not intended to be delivered as a stand alone product but as a building block for an application processor with embedded markets such as thin client, handheld devices, networking, storage, remote access servers, etc.

Arca2 CPU core is a Harvard cache architecture that is targeted at multiprogramming applications where full memory management, high performance and low power consumption are all important. In addition to the five-stage pipeline CPU, the core integrates a full featured MMU with separated 32 entry instruction TLB and data TLB, separated virtual tag instruction cache and data cache each with 8KBytes size, and a write buffer which greatly alleviate the memory latency.

Arca2 CPU core includes a debug module which provides a powerful mechanism for both hardware and software debugging. Hardware instruction and data breakpoints are provided. Through a JTAG interface, a software debugger could connect to the target processor without the need for extra hardware like serial or ethernet port.

Arca2 CPU core also integrates a performance monitor that could monitor a variety of performance events. This provides an efficient way for application performance tuning, benchmark evaluation and compiler optimization.

Arca2 CPU core interfaces to the rest of system through a unified Bus Interface Unit (BIU). The BIU could easily be connected to some on chip SOC bus, for example, the OCS(On Chip System) bus designed by ARCA Technology Corporation, or AMBA AHB bus by ARM.

1.2 Block Diagram

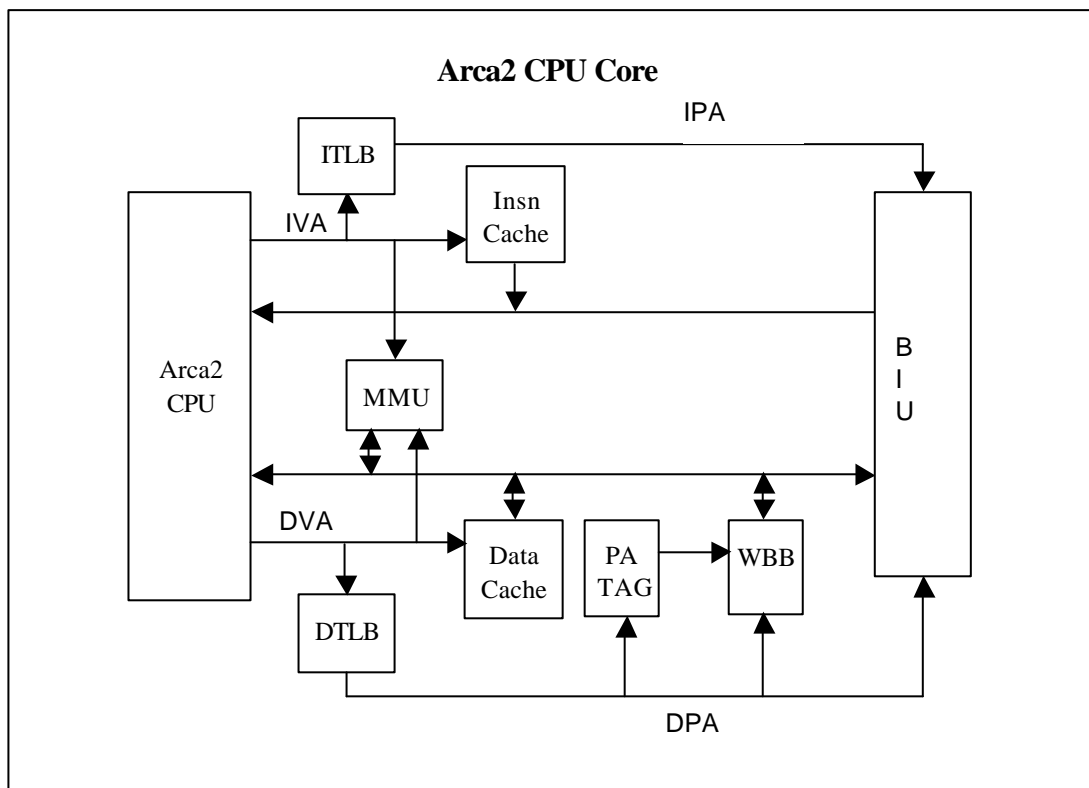


Figure 1-1 Arca2 CPU core Block Diagram

The block diagram of Arca2 CPU core is shown in Figure 1-1.

1.3 Features

The key features of Arca2 CPU core are listed in Table 1-1.

Table 1-1 Arca2 CPU Core Features

Item	Features
Arca2 CPU	<ul style="list-style-type: none"> • Arca version 2 architecture, 32-bit Arca instruction set. • 32 32-bit general registers • 5-stage pipeline • Interlocked implementation • Virtual address space: 4 G-Bytes
Memory Manager Unit (MMU)	<ul style="list-style-type: none"> • 4 G-Bytes of address space, divided into 5 partition spaces • Full associative 32-entry instruction TLB (ITLB) and 32-entry data TLB (DTLB), with round robin replacement algorithm • Four different page size: 4KB, 16KB, 1MB and 16MB in any entry • Support entry lock • Translate 32-bit virtual address to 32-bit physical address • Space identifier ASID: 8 bits, 256 virtual address spaces
Data Cache	<ul style="list-style-type: none"> • 8K-Byte, physically-indexed, virtually-tagged • Hardware resolve alias issue • 32-way set associative: 8 sets with each set containing 32 ways • Each way contains 32 bytes (one cache line) • Round robin replacement algorithm • Write-back, write-through • 4-word deep write buffer • Support lock, allocate operations
Instruction Cache	<ul style="list-style-type: none"> • 8K-Byte, physically-indexed, virtually-tagged • 32-way set associative: 8 sets with each set containing 32 ways • Each way contains 32 bytes (one cache line) • Round robin replacement algorithm • Support lock operation
Debug	<ul style="list-style-type: none"> • JTAG interface to host machine • ASID match • Two instruction or one maskable instruction address breakpoint • Two data or one maskable data address breakpoint • One data store result breakpoint • Software break • Asynchronous break from host machine • Asynchronous boot from host machine
Performance Monitor (PMON)	<ul style="list-style-type: none"> • One 32-bits internal clock counter • Two 32-bits signal counter, each of which can be set to count 1 of 15 signals • Count overflow interrupt

2 Arca2 CPU

2.1 Overview

Arca2 implementation uses a 5-stage pipeline design. The five stages are:

- IF - instruction fetch, fetch instruction from ICache or External Memory
- ID - instruction decode and GRF (general register file) read
- EX - instruction execution like addition, shift or the first part of multiply
- MA - memory access or the second part of multiply
- WB - write back result to GRF

Figure 2-1 shows general pipeline.

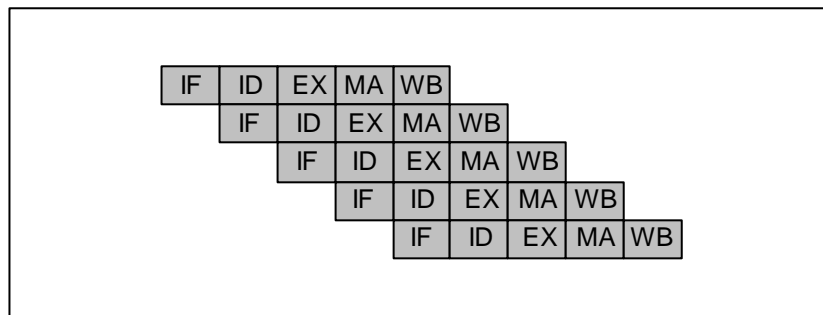
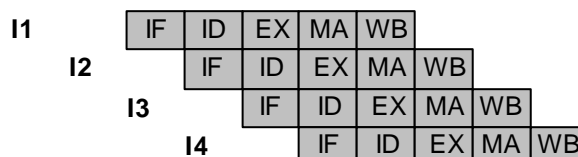


Figure 2-1 General pipeline

Arca2 pipeline will automatically interlock when a data dependence is detected by pipeline control. The interlocked implementation allows software to function identically across different implementations without concern for pipeline effects.

2.2 Pipeline for One-Pass and Multi-Pass Instructions

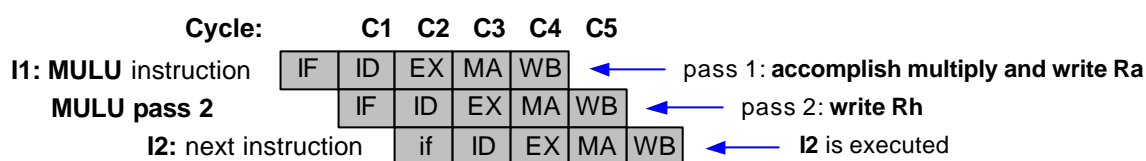
Most of Arca instructions are implemented to occupy one pipeline pass. For these instructions it seems that one instruction is executed within one cycle, which ensures high performance. It is illustrated by figure below, where all I1, I2, I3 and I4 instructions occupied one pipeline pass.



Besides, a few Arca instructions are implemented to occupy more than one pipeline pass, and thus multiple cycles are needed to be executed, such as MULU with high 32-bit result, ITLB, ICACHE and SLEEP.

2.2.1 MULU with high 32-bit result

MULU ($MULU\ Rh:Ra, Rb, Rc$) is special in that the execution could be 1 or 2 pipeline passes. If Rh is R0, i.e., the higher 32-bit result is not required, one pass is needed, otherwise it needs another pass to write the higher 32-bit result. The figure below illustrates the execution process of MULU with Rh not equal to R0:

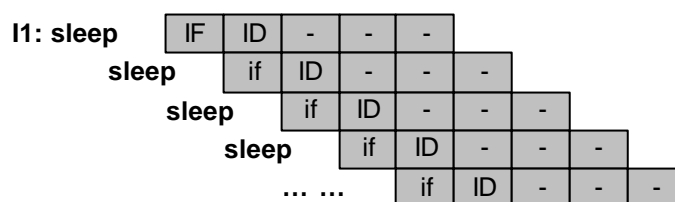


2.2.2 ITLB/ICACHE

ITLB is implemented as a two-pass instruction and ICACHE instruction is a four-pass one.

2.2.3 SLEEP

SLEEP is implemented as an 'infinite' pass instruction and the next instruction will not be executed until an interrupt or a wakeup signal breaks the execution of the SLEEP



2.3 Hazard & Forwarding

2.3.1 Forwarding method

Most Arca instructions carry out an operation with such a pattern: $R_a = R_b \text{ op } R_c$. In each cycle, there may be five instructions executed in the pipeline. If an operand that the current instruction needs to read is just the one that a preceding instruction will write, we called WR relation for short, the data dependency hazard occurs. Bypass or forwarding technique is used to solve this kind of hazard.

When WR occurs, a result data of previous instruction is needed by current instruction before the data is written to GRF. With forwarding technique, the data is forwarded directly to current instruction when it is on the way to GRF, instead of waiting it write to GRF then read from GRF. So the forwarding technique prevents a big lost in CPU performance. Arca2 CPU employs this technique in all possible circumstance that solves most of the data dependency hazard.

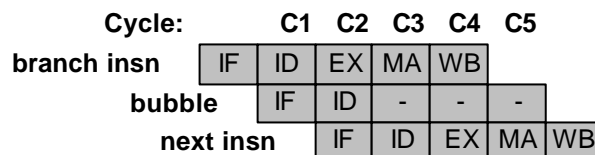
2.3.2 Hazard

Arca has 3 kinds of hazard. The first one is **control hazard** caused by branch instruction or exception acknowledgement. The second one is **data hazard** caused by WR relationship. The last one is **structure hazard** caused by multiplier resource contention.

2.3.2.1 Control hazard

1. Jump and branch instruction

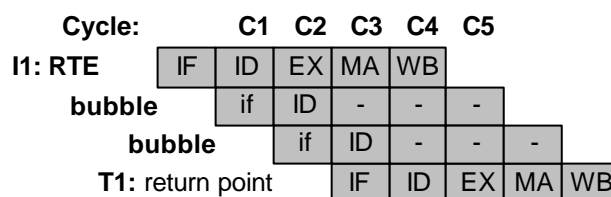
When jump or branch instruction executed, there is always one bubble between the jump/branch instruction and the next one. That is to say, there always exists 1 cycle penalty for Arca2 CPU to execute a jump or a branch instruction (refer to the figure below).



Note that for BCC/BCCI instructions, there is only 1 cycle penalty no matter whether the branch is taken or not.

2. RTE instruction

When RTE instruction is executed, there are always two bubbles between RTE and the next instruction T1, which is in the instruction flow before exception happen.



3. Exception

All kinds of exception could cause control hazards, see Figure 2-2.

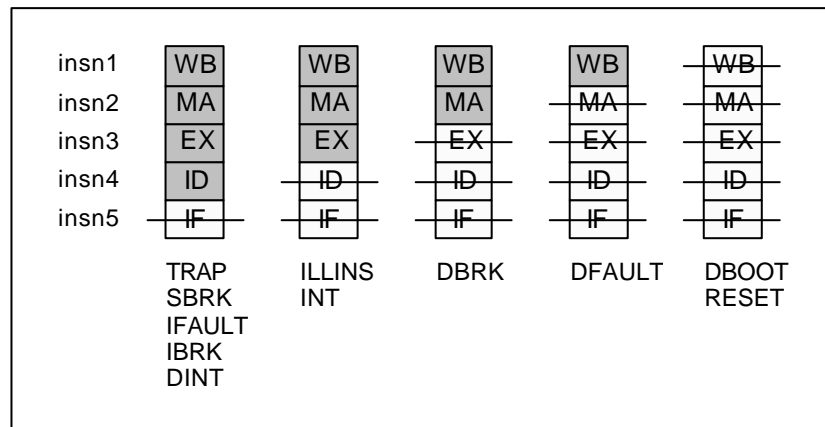


Figure 2-2 Exception hazard

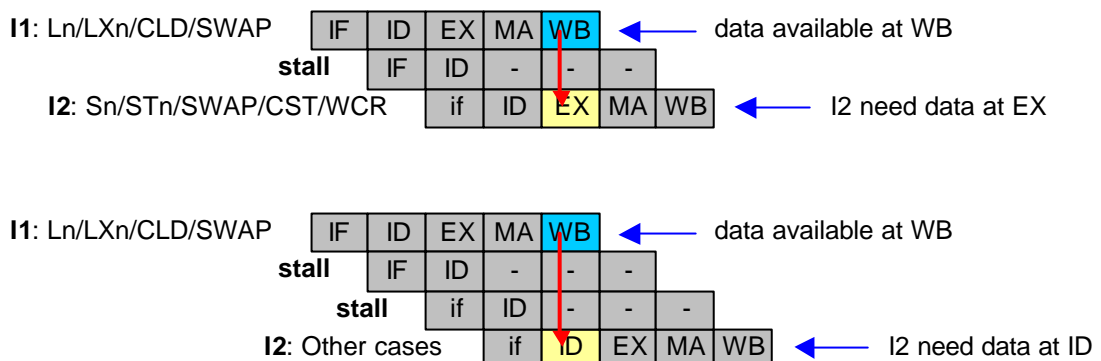
Note that the deep gray area in above figure means that instruction could flow forward without any effect from exception, while the dash area denotes that these instructions' pipeline should be cut down.

2.3.2.2 Data hazard (Stall)

Data hazard is generally caused by WR relationship between two adjacent instructions. If the first instruction produces a data in a later stage and the next instruction uses it in an earlier stage, the data consumer has to wait till the data is available. The instruction execution pipeline is then stalled. The data hazard caused by the different type instruction will be discussed in the following:

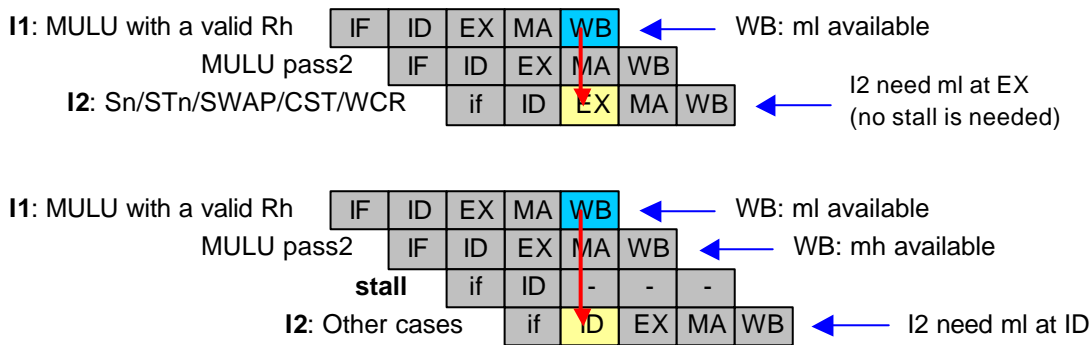
1. Read after Load

Since the result of a load instruction is ready at the end of MA stage, 1 stall is needed for the next instruction if it is one of the store, SWAP, CST or WCR instruction that uses the result as the data to be stored. 2 stalls are needed for the next instruction if it is one of other instructions that uses the result or other using cases.



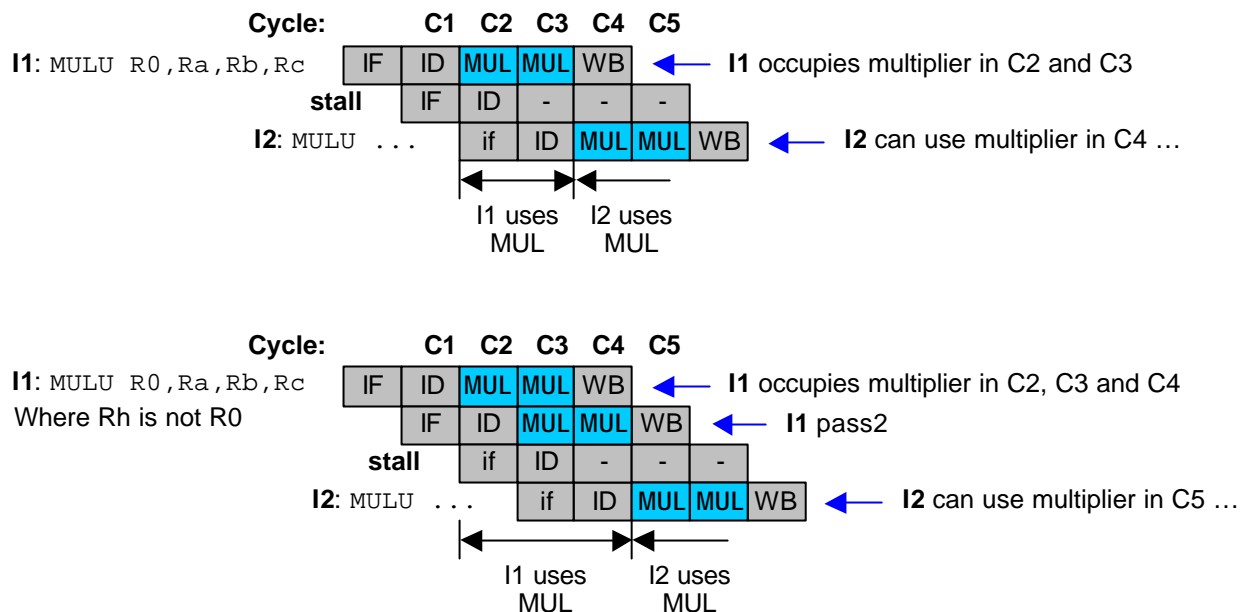
2. Read after MULU

Similar to a load instruction, MULU gets its result (ml for low 32-bit and mh for high 32-bit) at the end of MA stage and is available at WB stage. The stall case when 'mh' is not required is just the same as read after load. When the higher 32-bit result (mh) is required, MULU lasts two pipeline passes, where ml is available at WB stage of pass 1, and mh is of pass 2. The stalls needed between mh producing and using is also the same as the read after load case. The figures below illustrates the pipeline sequence for two-pass MULU and its following instruction that using MULU's ml:



2.3.2.3 Structure hazard

Structure hazard in Arca is caused by multiplier resource contention. One MULU instruction will occupy multiplier resource for two or three consecutive cycles, when a MULU immediately followed by another MULU, the resource contention occurs, which requires one bubble being inserted between two consecutive MULU instructions. The figure below explains multiplier resource contention:



Notes structure hazards caused by cache/memory resource contention are described in "Chapter 6 Cache"

2.4 Cycles for Arca2 instruction executions and stalls

Table 2-1 summarizes static cycle consumption for all instructions and IU states and Table 2-2 summarizes dynamic cycle consumption for all cases that cause stall.

Table 2-1 Instruction and special IU states cycles

IU state	Instruction	Cycles	Description
Power-on RESET	-	32	1. Wait peripheral devices to be initialized 2. Clear GRF
Exception	-	5	When an exception/interrupt is detected and accepted, 5 extra cycles are needed before execute the first instruction in exception/interrupt routine
SLEEP	SLEEP	-	SLEEP instruction will be repeatedly executed until an interrupt or a wake up signal
NORMAL STATE	ICACHE	4	
	ITLB	2	
	MULU Rh,Ra,Rb,Rc	2	When Rh is not R0
	Others	1	

Table 2-2 Stall conditions and cycles

Instruction May Cause Stalls	Stall Condition	Stall Cycles
BEQ BNE BLT BLTU BGE BGEU BEQI BNEI BEQUI BNEUI BLTI BLTUI BGEI BGEUI J JA	Always	1
RTE	Always	2
L8 L8U L16 L16U L32 LX8 LX8U LX16 LX16U LX32 LX16S LX16SU LX32S CLD SWAP	The loaded data is used as a stored data ⁽¹⁾ in next instruction	1
	The loaded data is used by next instruction as other than a stored data	2
	The next instruction consume one cycle and the loaded data is used by the next of the next instruction as other than a stored data	1
MULU R0,Ra,Rb,Rc	Same as the load instruction cases, just replace the "loaded data" by "multiply result"	1/2
MULU Rh,Ra,Rb,Rc (where Rh is not R0)	Rh data dependence is just like above, just replace the "multiply result" by "multiply high 32-bit result"	1/2
	The multiply low 32-bit result is used by the next instruction as other than a stored data	1
MULU	If next instruction is a MULU again	1

Note

(1) "**stored data**" means the data to be stored (or Ra) of the instruction CST SWAP WCR S8 S16 S32 SX8 SX16 SX32 SX16S SX32S.

Please note when there are multiple stall conditions meet simultaneously, the longest stall takes place.

3 Exception Model

3.1 Overview

Arca2 CPU core provides a simple and efficient way to organize and handle exceptions. By providing an 8-entry vector table with each entry corresponding to one or more exception types, Arca2 can switch to the top exception routine conveniently and efficiently. The top exception routine may consult the exception cause register to further determine the specific exception service.

3.2 Exception Types

There is a variety of resources that can trigger an exception to CPU. Figure 3-1 illustrates the various resources that will request exception services from CPU.

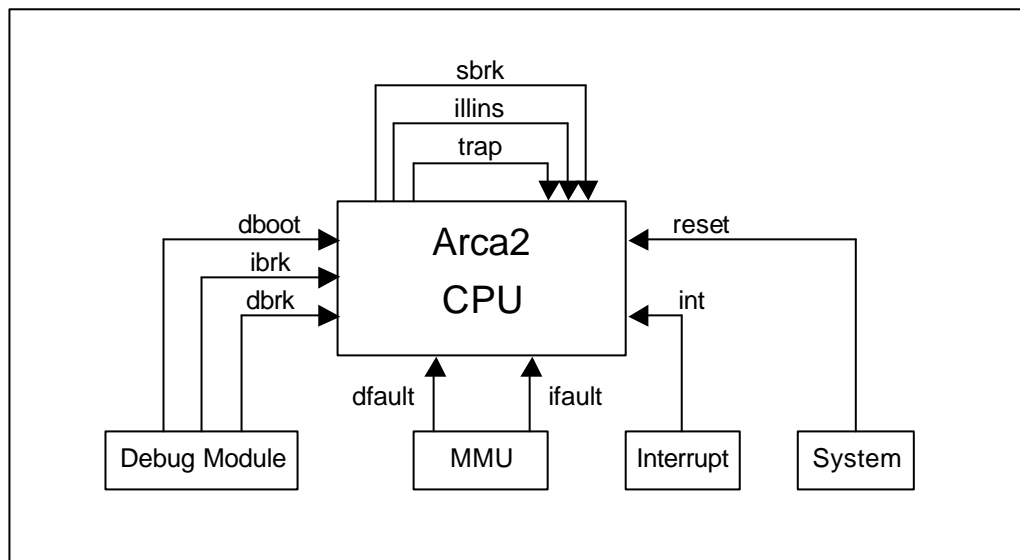


Figure 3-1 Exceptions and Exception Resources

Debug and MMU are modules inside the CPU core, they will issue exceptions to CPU for debug and memory access event. Events such as reset and hardware interrupt are issued by modules outside the CPU core such as an interrupt or system controller. CPU itself will generate exceptions when executing a special instruction.

There are total 10 types of exceptions supported by Arca2 CPU Core as illustrated in Figure 3-1:

- **RESET:** Reset exception request by a system controller outside of the CPU core. Reset exception can be induced by a power-on reset and manual reset from external input pin or a watchdog time-out reset.
- **INT:** Interrupt exception issued by an interrupt controller outside of the CPU core.
- **Dfault:** Data access fault exception occurred when a data access request by CPU couldn't be satisfied by memory subsystem.

- **Ifault:** Instruction fetch fault occurred when an instruction access request by CPU couldn't be satisfied by memory subsystem.
- **Illins:** Illegal instruction caused by executing a reserved instruction or there is a privilege violation in executing the instruction.
- **Trap:** the OS trap mechanism caused by executing a trap instruction.
- **Sbrk:** a software breakpoint instruction caused by executing a SBRK instruction when debug module support is enabled by SR.DE bit.
- **ASYNBRK:** Debug asynchronous break from host machine.
- **IBRK:** Instruction break caused when an instruction fetch address appeared in the instruction address bus matches the set in the debug module's instruction breakpoint register. The asynchronous
- **DBRK:** Data break caused when a data access address appeared in the data address bus or a stored data in data bus matches the set in debug module's data breakpoint register.
- **DBOOT:** Debug Bootstrap. Issued by the debug module when it receives a command from the extended JTAG ports by the host machine debugger.

The exceptions supported by Arca2 CPU core can be classified into 2 categories: normal exceptions and debug exceptions. Normal exceptions include **Reset, Dfault, Ifault, Illins, Trap, INT**, and debug exceptions include **DBOOT, IBRK, DBRK, SBRK** and **ASYNBRK**. CPU uses ESR, EPC for normal exception and DSR, DPC for debug exception.

3.3 Exception Priorities

More than one exception could request CPU's attention simultaneously. When this situation occurs, the exception that has the highest priority will be accepted by CPU. The exception priorities are fixed as illustrated by the table below:

Table 3-1 Arca Exception Priorities

Exception Kinds	Exception Events	Exception Priorities
DBoot	Debug Bootstrap	0 (highest)
Reset	Power-on/Manual Reset	1
DFAULT	Data Access Fault	2
DBRK	Debug Data Breakpoint	3
ILLINS	Reserved Instruction or Privilege Violation	4
TRAP/SBRK	TRAP or SBRK Instruction	5
IBRK/ ASYNBRK	Debug Instruction Breakpoint or asynchronous break from host machine	6
IFault	Instruction Fetch Fault	7
INT	Interrupt	8 (lowest)

3.4 Exception Vector Table

Arca2 CPU core uses 8word memory space to hold exception vectors. The vector table is shown in Table 3-2:

Table 3-2 Arca Exception Vector Table

Vector Number	Vector Offset	Exceptions
0	H'00	RESET, DBOOT
1	H'04	ILLINS
2	H'08	IBRK, DBRK, SBRK, ASYNBRK
3	H'0c	Reserved
4	H'10	INT
5	H'14	TRAP
6	H'18	DFAULT, IFAULT
7	H'1c	Reserved

The base address of the vector table base for DBOOT exception is always at H'EC000000.

When Arca2 CPU core is set to host mode by a debugger running in a host machine, the exception vector table base is fixed at H'EC000000.

For other cases, exception vector table is placed on the boundary of 64M memory page of P1 area, which is decided by SR.VB bits (refer to Status Register description).

$$\text{Base address} = \{3B'100, \text{SR.VB}, 26B'0\}$$

Since SR.VR is initialized to 3B'000, so the base address for power-on reset without DBOOT is always H'80000000.

3.5 Exception Cause

In Table 3-2, there are several vector numbers that contain more than one exception type. Vector number 0 is the exception handler entry address for **RESET** and **DBOOT**; vector number 2 is the exception handler entry address for **IBRK**, **DBRK**, **SBRK** and **ASYNBRK**; vector number 6 is the exception handler entry address for **DFAULT** and **IFault**. This arrangement limits the vector table size to an 8 words size. A small vector table could be locked into data cache line, thereby improve the performance for exception handling.

Since more than one exception types share one exception vector table entry, a mechanism is needed for differentiate the exception type by the exception handler routine. The cause register of MMU and Debug module serves this purpose. The interrupt controller outside the CPU core will provide a similar register. The immediate number operand in TRAP instruction provides more information for its exception handler.

Table 3-3 Exception Cause

Exception Type	Exception Cause Register	Cause Register Set by	Instruction to Read Cause Register
TRAP	General Register	The trap number can be passed to a general register via TRAP instruction	
DFAULT IFault	MCR.cause	MMU	CLD Ra,#MMU,#MCR
INT	A register in INTC	INTC	load from the memory-mapped location
IBRK DBRK SBRK ASYNBRK	DCR.cause	DEBUG	CLD Ra,#DEBUG,#DCR

3.6 Control Register

The **SR**, **ESR**, **EPC**, **DSR** and **DPC** are control registers of IU. They play the key roles of the exception model. Here we give the detailed description.

3.6.1 Status Register (SR)

Bit:	31	6	5	4	3	2	1	0
Read:		VB			SM	DS	DE	IE
Write:								
Reset:	0	0	0	0	1	0	0	0

Bits 31~7 reserved, these bits are always read as 0 and written are ignored.

- **IE (Interrupt Enable):** When it is cleared, interrupt is disabled.
- **DE (Debug Enable):** When it is cleared, no Debug Exceptions are to be accepted exception DBOOT. Debug bootstrap can't be disabled by this bit.
- **DS (Debug State):** 1 indicates an exception is a debug exception (which include debug bootstrap), 0 for other exceptions. RTE restore PC/SR register from DPC/DSR when it is 1, from EPC/ESR when it is 0.
- **SM (Supervisor Mode):** 1 for Supervisor mode, 0 for User mode. Write to this bit by WCR instruction is ignored. Program should use RTE to switch from supervise mode to user mode by first clear corresponding bit in ESR or DSR register.
- **VB (Vector Base):** form bit 28~26 of the base address of Vector Table (the highest 3 bits are 100, pointing to P1 area), when it is neither a DBOOT exception nor in host mode.

3.6.2 Spot-saving Register

- **ESR:** used to save the current status register for none debug exceptions

Bit:	31	6	5	4	3	2	1	0
Read:		VB			SM	DS	DE	IE
Write:								
Reset:	0	undefined						

- **EPC:** used to save none debug exception return address.

Bit:	31	2	1	0
Read:	PC[31:2]			0
Write:				
Reset:	undefined	0		

- **DSR:** used to save the current status register for debug exceptions

Bit:	31	6	5	4	3	2	1	0
Read:		VB			SM	DS	DE	IE
Write:								

Reset: 0 undefined

- **DPC**: used to save debug exception return address.

Bit:	31	2	1	0
Read:	PC[31:2]			0
Write:				0
Reset:	undefined			0

3.7 Exception Acknowledgement Process

It takes several cycles for Arca2 CPU to switch from the current program flow to the exception routine.

The CPU exception acknowledgement process fulfills the following jobs:

- Save the Status Register (**SR**) to **ESR** for none debug exception or to **DSR** for debug exception.
- Compute the return address and save it to **EPC** for none debug exception or to **DPC** for debug exception.
- Form the vector entry address based on the exception type and vector base. Load the exception handler start address from the vector entry. Fetch the first exception handler instruction from the loaded exception handler start address.

If the process is in **SLEEP** state, put it into normal state.

- Enter into privilege mode by setting **SR.SM**
- Clearing **SR.IE** to disable interrupt exceptions
- For debug exceptions, clearing **SR.DE** to disable additional debug exceptions.
- Set **SR.DS** to 0 for none debug exceptions, or set **SR.DS** to 1 for debug exception.

3.8 Return from Exception Routine

Return from exception routine is implemented by the instruction **RTE**, which runs in privilege mode and fulfills the following jobs:

- Restore previous system status by copying **ESR** if **SR.DS** is 0, or **DSR** if **SR.DS** is 1, into **SR**.
- Restore execution of previous program by jumping to the return address stored in **EPC** if **SR.DS** is 0, or **DPC** if **SR.DS** is 1.

To ensure no interrupt exception is acknowledged and thus clobber the content in **ESR/EPC**, **SR.IE** must be cleared before execution of **RTE** if **SR.DS** is 0. Similar, for not clobbering **DSR/DPC**, **SR.DE** must be cleared before execution of **RTE** if **SR.DS** is 1.

4 Core Configuration

4.1 Overview

In addition to CPU, Arca2 CPU core includes other modules such as MMU, data TLB (DTLB), instruction TLB (ITLB), data cache, instruction cache, debug module and performance monitor (PMON). Later versions of the CPU core may add more modules to expand functionality. Arca architecture provides a uniform and extensible way to manage these modules with the instructions listed below. These instructions provide a consistent way to exchange values between CPU register file and control registers of a specific module, and to expand module specific operations.

CLD	Ra, ID, S10
CST	Ra, ID, S10
ITLB	CMD, Rb
DTLB	CMD, Rb
ICACHE	CMD, Rb, S10
DCACHE	CMD, Rb, S10

4.2 Configuration Instructions

4.2.1 CLD/CST instruction



These two instructions exchange data between CPU register file and a module's control register. CLD loads a 32 bits data into CPU register Ra from the control register specified by S10 field in the module specified by ID field. CST writes the data in Ra to the control register specified by S10 field in the module specified by ID field.

- Ra: CPU register. For CLD instruction, this is the destination register. For CST instruction, this is the source register.
- ID: module indentify code. Arca2 CPU core defines 3 module as illustrated in the following table:

Table 4-1 Module Identification Number

Module Name	MMU	PMON	DEBUG
ID	000	001	011

- S10: The control register number inside module ID. The control number can be **0 ~ 511**. The access right to a specific control register is defined by the module itself. For example, MCR and CCR register in MMU module can't be accessed in user mode. When the access right violation happens, a default exception will be induced to CPU by MMU module.

The control registers defined in module MMU are as below. See section 5 for the detailed description of these control registers.

Table 4-2 Control Registers in Module MMU

CR Name	MCR	TTB	MEA	CED	ASI	CCR
Number	000	001	010	100	011	101

The control registers defined in module Debug are as below. See section 7 for the detailed description of these control registers.

Table 4-3 CR in Module Debug

CR Name	DBG_CR	DBG_IA0	DBG_IA1	DBG_DA0	DBG_DA1	DBG_DD0	DBG_ASID
Number	000	001	010	011	100	101	110

The control registers defined in module PMON are as below. See section 8 for the detailed description of these control registers.

Table 4-4 Control registers in Module PMON

CR Name	PMC	CTR	MOR0	MOR1
Number	000	001	010	011

4.2.2 ITLB/DTLB/ICACHE/DCACHE instructions

ITLB CMD, Rb

Op: 000000	0CMD	Rb	Ext: 001101	Don't care
------------	------	----	-------------	------------

DTLB CMD, Rb

Op: 000000	0CMD	Rb	Ext: 001001	Don't care
------------	------	----	-------------	------------

ICACHE CMD, Rb, S10

Op: 000000	0CMD	Rb	Ext: 000101	S10
------------	------	----	-------------	-----

DCACHE CMD, Rb, S10

Op: 000000	0CMD	Rb	Ext: 000001	S10
------------	------	----	-------------	-----

These instructions are used for special operations applied on Arca embedded memory. The specific position on RAM is defined by the virtual address: $[Rb+S10<<2]$ while the specific operation is defined by each kind of RAM's through 'CMD'. Note some of CMDs may be executed in User Mode while some of them may not. When executing a privileged CMD in User Mode or the CMD does not exist, a default exception request will be asserted.

4.2.3 CMD in ITLB/DTLB/ICACHE/DCACHE

The module CMD (4-bits) is defined as below:

Table 4-5 Module CMD Definitions

CMD Number	RAM Name			
	ITLB	DTLB	ICache	DCache
0000			Prefetch	Prefetch
0001	Discard	Discard	Discard	Discard
0010	Read	Read		Write-back
0011	Write	Write		Flush
0100				Allocate
0101				
0110				ALock
0111	PLock	PLock	PLock	PLock
1000				
1001				Flush-buffer
1010				
1011				
1100				
1101				
1110	Unlock	Unlock	Unlock	Unlock
1111	Invalidate	Invalidate	Invalidate	Invalidate

Notes

- (1) For DTLB/ITLB all operations are only valid in privileged mode, for ICache/DCache, CMD[2:1] specifies the access right, i.e., '11' for privileged mode only, others for both modes,
- (2) For flush-buffer, invalidate and unlock commands, the address specified by the instruction is ignored. CMD[3]=1 specifies this

The commands are explained as below:

- **Prefetch:** prefetch data or instructions into cache line, during prefetch process, ICache/DCache should not freeze the IU-pipeline.
- **Discard:** clear V-bit of the specified TLB entry or cache line for the specified TLB or cache.
- **Write-back:** write back data of the specified cache line if D-bit is set, then clear D-bit.
- **Flush:** this is the combination of **discard** and **write-back**, that is, write back the cache line if it is dirty and clear D and V bits.
- **Read:** read the specified TLB entry, put the PPN and attribute bits into CED register
- **Write:** write a TLB entry, PPN and attribute bits from CED register, VPN from the specified address
- **Allocate:** allocate a cache line for the virtual address, that is, only fill the TAG into a cache line but does not care the data.
- **Flush-buffer:** flush the write back buffer in DCache.
- **ALock:** for DCache only, allocate and lock a cache line.
- **PLock:** for ITLB/DTLB, load and lock an entry, for ICache/DCache, load and lock a cache line.
- **Unlock:** unlock all locked TLB entry or cache line for the specified TLB or cache.
- **Invalidate:** clear V-bits for entries those not been locked in the specified TLB or cache.

5 Memory Manage Unit

5.1 Overview

MMU serves as a powerful manager to make efficient use of physical memory. To accelerate translating virtual memory to physical memory, Arca2 CPU core uses both an instruction Translation Look-aside Buffer (ITLB) and a data Translation Look-aside Buffer (DTLB) to cache the latest translation. Arca2 CPU core supports four page sizes: 4KB, 16KB, 1MB, and 16MB. MMU also controls virtual memory access permission for different processor mode: privileged mode and user mode.

5.1.1 Features

- MMU equips Translation Look-aside Buffer (TLB) for both instruction fetch and data access to accelerate virtual to physical address translation. Each TLB holds 32 entries and is full associative.
- Use round-robin replacement method and support lock function to lock critical entries in DTLB or ITLB.
- Virtual Address translation uses the paging system and supports four page sizes: 4KB, 16KB, 1MB and 16MB bytes.
- Virtual Address map to physical address space directly when disable paging system.
- MMU checks the memory access permission in different processor modes to provide access protection.
- MMU issues the exception request to CPU when the instruction fetch or data access encounters a fault. It also saves spot information such as fault address and fault cause to be referenced by the exception routine.

5.2 Register Configuration

The Arca2 CPU core provides several 32-bit MMU and cache control registers, which determine the operation of MMU and Cache. A brief description of the registers is provided below.

Data is written to and read from the MMU registers using the Arca2 **CST/CLD** instructions.

The **MMU Control Register** holds the control signal bits and exception cause bits, which determine the operation of MMU.

The **Translation Table Base Register** holds the base physical address of the translation table maintained in main memory.

The **MMU Exception Address Register** holds the virtual address where the exceptions occur.

The **Address Space Identifier Register** holds process ID number.

The **Configure Exchange Data Register** holds the data for read, write or lock TLB operation.

The **Cache Control Register** holds control signal bits that determine the operation of Cache. The detail description is in Cache spec.

Table 5-1 MMU Registers

Name	Full Name	R/W	Initial value when power on	Access Size	#ID	#CR
MCR	MMU Control Register	R/W	H'00000000	32	000	000
TTB	Translation Table Base Register	R/W	Undefined	32	000	001
MEA	MMU Exception Address Register	R/W	Undefined	32	000	010
CED	Configure Exchange Data Register	R/W	Undefined	32	000	100
ASI	Address Space Identifier Register	R/W	Undefined	32	000	011
CCR	Cache Control Register	R/W	H'00000000	32	000	101

5.2.1 Register Descriptions

5.2.1.1 MMU Control Register (MCR)

#ID=000 #CR=000

Bit:	31	30	29	28	27	26	25	24
Read:	CAUSE							
Write:								
Reset:	0	0	0	0	0	0	0	0

Bit:	23	22	21	20	19	18	17	16
Read:								
Write:								
Reset:	0	0	0	0	0	0	0	0

Bit:	15	14	13	12	11	10	9	8
Read:								
Write:								
Reset:	0	0	0	0	0	0	0	0

Bit:	7	6	5	4	3	2	1	0
Read:								ATE
Write:								
Reset:	0	0	0	0	0	0	0	0

Bit 24 ~ 1: Reserved bits, ignored in write operation, always 0 in read operation.

- **ATE:** address translation enabled/disabled bit.
0: address translation disabled.
1: address translation enabled.
- **CAUSE:** Exception causes bits.
Bit31: TLB miss or not.
Bit30: Address error.
Bit29: Reserved bit, ignored in write operation, always 0 in read operation.
Bit28: Initial write.
Bit27: Exception occurs in instruction fetch or data access 0: data; 1: instruction.
Bit26: Exception occurs in LOAD or STORE operations. 0: LOAD, 1: STORE(include swap operation).
Bit25: Exception occurs in core configure instruction.

The valid cause patterns are listed in following table.

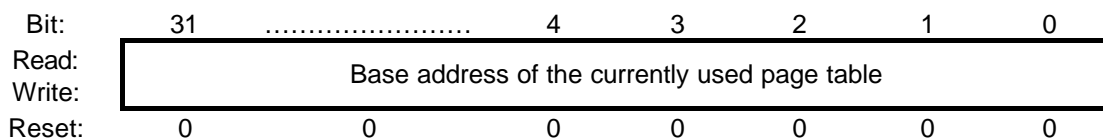
CAUSE	Description
1000_100	ITLB miss when fetch instruction.
1000_101	ITLB miss when use I-cache instruction.
0100_100	address error when fetch instruction.
0100_101	address error when use I-cache or ITLB instruction.
0000_101	illegal I-cache or ITLB operation.
1000_000	DTLB miss when read access
1000_010	DTLB miss when store or swap access

CAUSE	Description
1000_001	DTLB miss when D-cache instruction
0100_000	address error when read access
0100_010	address error when store or swap access
0100_001	address error when use D-cache or DTLB instruction
0001_010	initial write when store or swap access
0000_001	illegal cache or TLB configure operation or CLD/CST operation.
0000_000	no exception, initial value when power-on reset.

5.2.1.2 Translation Table Base (TTB)

Point to the base address of current page table. This register is managed by software.

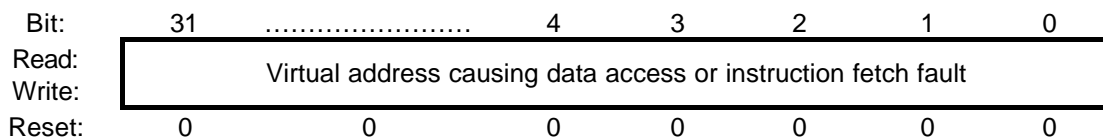
#ID=000 #CR=001



5.2.1.3 MMU Exception Address (MEA)

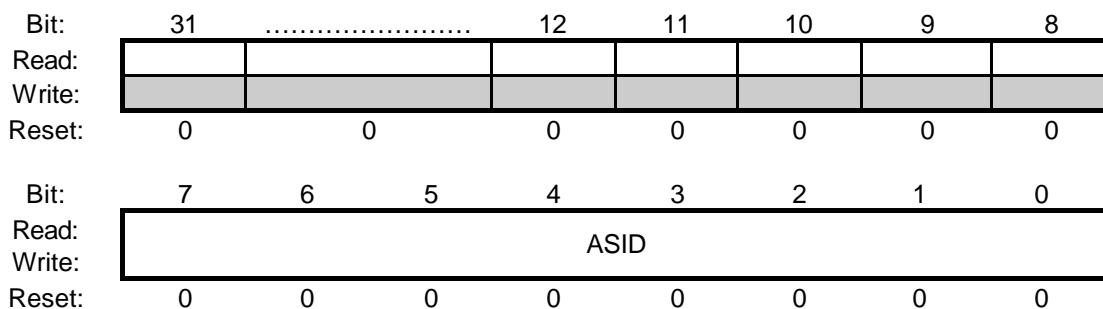
When MMU exception occurs, the virtual address that induces exception is set into this register by hardware. The contents of this register can be changed by software.

#ID=000 #CR=010



5.2.1.4 Address Space Identifier (ASI)

#ID=000 #CR=011



Bit 31 ~ Bit 8: Reserved bits, ignored in write operation, always 0 in read operation.

- **ASID:** Address space identifier. ASID indicates current process, which is regarded as expansion of virtual memory.

5.2.1.5 Configure Exchange Data Register (CED)

#ID=000 # CR =100

Bit:	31	30	29	28	27	26	25	24
Read:	PPN							
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	23	22	21	20	19	18	17	16
Read:	PPN							
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	15	14	13	12	11	10	9	8
Read:	PPN							
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	7	6	5	4	3	2	1	0
Read:			SZ		D	B	C	M
Write:								
Reset:	0	0	0	0	0	0	0	0

Bit 11 ~ Bit6: Reserved bits, ignored in write operation, always 0 in read operation.

- PPN: Physical page number.
- SZ: 2 page size bits.
 - 00: 4KB page size
 - 01: 16KB page size
 - 10: 1MB page size
 - 11: 16MB page size
- D: Dirty bit.
Used only in DTLB. Ignored when write ITLB, undefined when read ITLB.
- B: Bufferable bit.
Used only in DTLB. Ignored when write ITLB, undefined when read ITLB.
- C: Cacheable bit.
- M: Read ITLB/DTLB miss bit.
Ignored when write ITLB/DTLB, set 1 when read ITLB/DTLB miss, clear 0 when read ITLB/DTLB hit. When M is 1 after read ITLB/DTLB, the other bits' value is undefined.

5.3 Memory Space

Arca2 CPU core supports a 32-bit physical address space, and can access a 4-Gbyte space, and use virtual memory system to logically expand the physical memory space of the processor, by translating addresses composed in a large virtual address space into the physical address space of the system. Arca2 CPU core has two modes to translate virtual address: paging system mode, and direct map mode.

5.3.1 Direct Map Virtual Address Space

When the MCR.ATE bit is reset to 0, the MMU address translation is in direct map mode. The virtual address space mapping is shown in Figure 5-1.

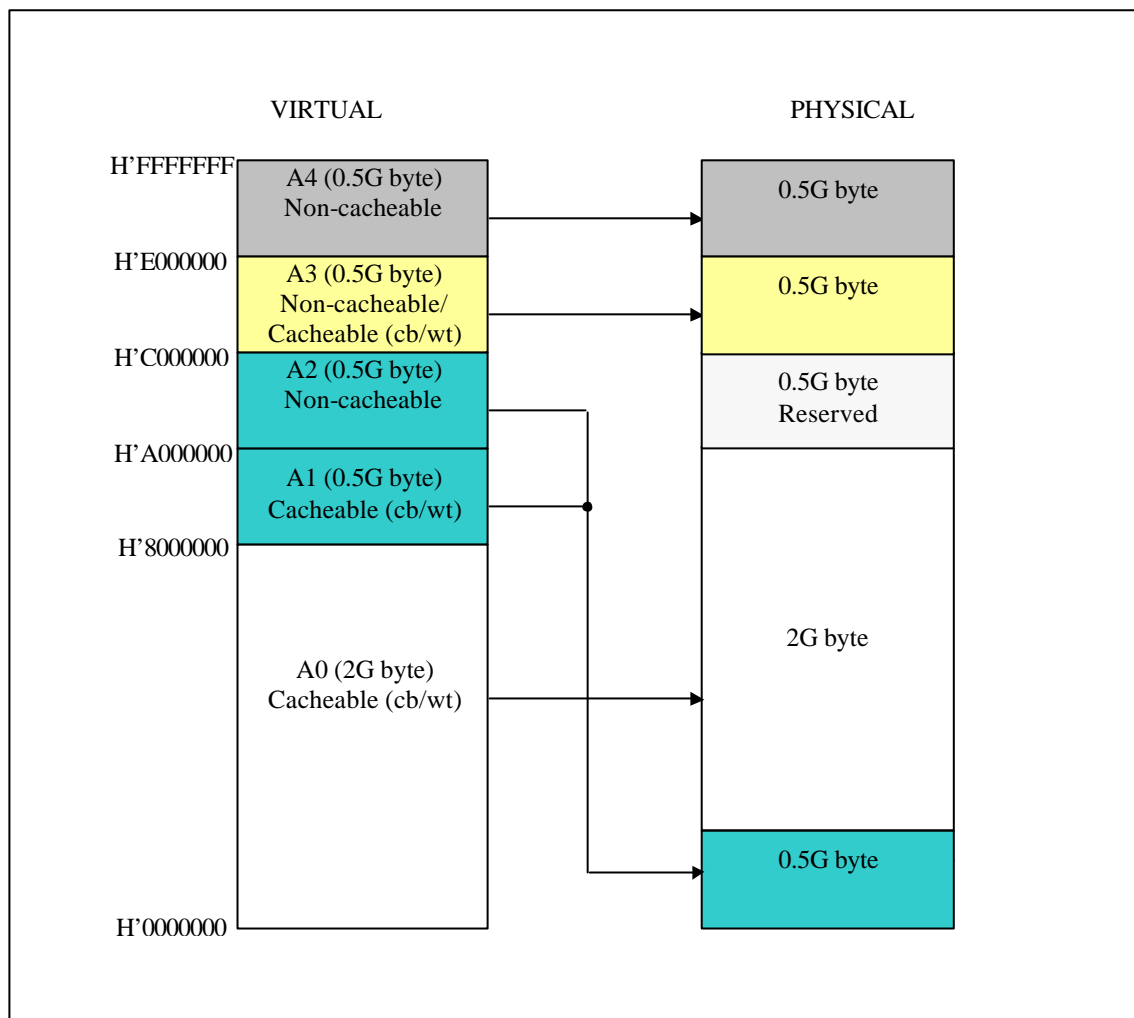


Figure 5-1 Direct Virtual Physical Address translation (MCR.ATE=0)

- **A0 Area:** The A0 areas can be accessed using the cache. When use cache, the write policy can be programmed to copy-back(cb) or write-through(wt). This area is always translated to linear 2GB region of the physical address space (from 'H20000000 to 'H9FFFFFFF) .

- **A1 Area:** The A1 area can be accessed using the cache, When use cache, the write policy can be programmed to copy-back(cb) or write-through(wt). This area is always translated to a linear 512MB region of the physical address space starting at physical address 0.
- **A2 Area:** The A2 area cannot be accessed using the cache. The write can be programmed to bufferable or unbufferable. This area is always translated to a linear 512MB region of the physical address space starting at physical address 0.
- **A3 Area:** The cacheable attribute of A3 area can be programmable (see cache spec), default is uncacheable. When use cache, the write policy can be programmed to copy-back(cb) or write-through(wt). When doesn't use cache, the write can be programmed to bufferable or unbufferable. This area is always translated to a linear 512MB region of physical address (from H'C0000000 to H'DFFFFFFF).
- **A4 Area:** This area cannot be accessed using the cache. The write is unbufferable. Some A4 area is mapped to on-chip I/O registers channels and some is mapped to JTAG memory. This area is always translated to a linear 512M region of physical address(from H'E0000000 to H'FFFFFFFF).

In user mode, the 2Gbyte of virtual address space from H'00000000 to H'7FFFFFFF(area A0) can be accessed. The 2 Gbytes of virtual address space from H'80000000 to H'FFFFFFFF cannot be accessed in user mode. Attempting to do so creates an exception named address error.

5.3.2 Virtual Address Space In Paging System

When the MCR.ATE bit is set to 1, MMU is in paging system mode. Arca2 CPU core uses 32-bit virtual addresses to access 4Gbyte virtual address space that is divided into several areas. Address space mapping is shown in Figure 5-2.

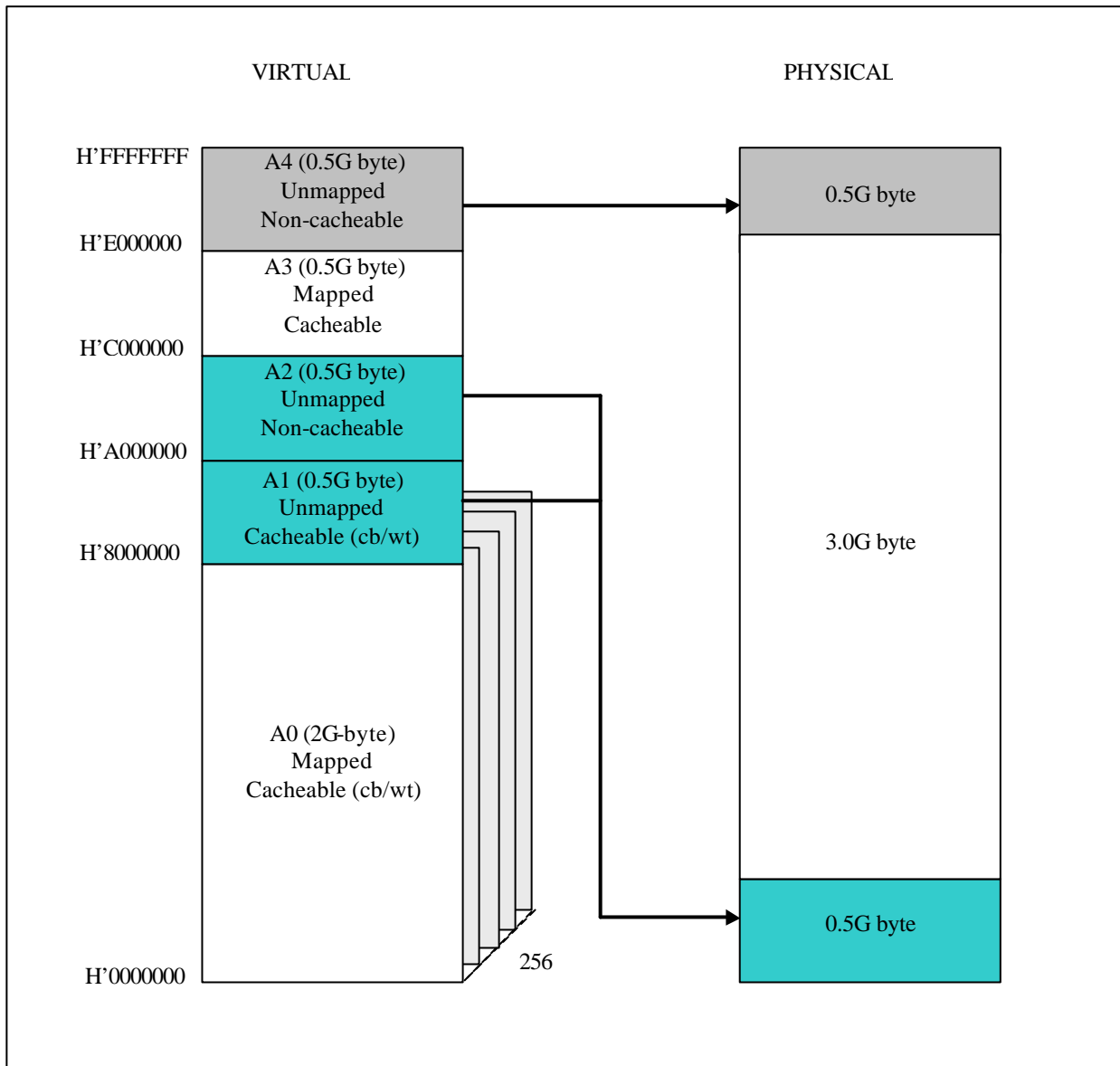


Figure 5-2 Paging Virtual Physical Address translation (MCR.ATE=1)

Setting the MCR.ATE bit to 1 enables the A0, and A3 areas of the address space in the Arca2 CPU core to be mapped onto any external memory space. By using 8-bit ASID address identifier, the A0 can be increased to a maximum of 256. A3 is a global area, all processes share it. This is called the virtual address space. Mapping from virtual address space to 32-bit physical memory space is carried out using the TLB.

- **A0,A3 areas:** These areas are mapped onto physical address space in page unites. In accordance with TLB information and CCR, these areas can be cacheable or noncacheable, bufferable or unbufferable.
- **A1 area:** This area is fixed to 512 MB physical address space starting from physical address 0, this area can be cached. When use cache, the write policy can be programmed to copy-back(cb) or write-through(wt).
- **A2 area:** This area is fixed to 512 MB physical address space starting from physical address 0, this area cannot be cached. The write can be programmed to bufferable or unbufferable.
- The A1 and A2 areas are not mapped by the address translation table, so the TLB is not used and no TLB exceptions like TLB misses occur. Initialization of MMU-related registers, exception process handling, and the like codes can be located in the A1 and A2 areas. Because the A1 area is cached, handlers that require high-speed processing are placed there.
- **A4 Area:** This area cannot be accessed using the cache. The write is unbufferable. Some A4 area is mapped to on-chip I/O registers channels and some is mapped to JTAG memory. This area is always translated to a linear 512M region of physical address(from H'E0000000 to H'FFFFFFFF).

In user mode, the 2Gbyte of virtual address space from H'00000000 to H'7FFFFFFF(A0) can be accessed. The 2 Gbytes of virtual address space from H'80000000 to H'FFFFFFFF cannot be accessed in user mode. Attempting to do so creates an address error.

5.4 Configuration of the TLB

The TLB caches address translation table information located in external memory. Figure 5-3 shows the DTLB and ITLB configuration. Both DTLB and ITLB are full associative with 32 entries.

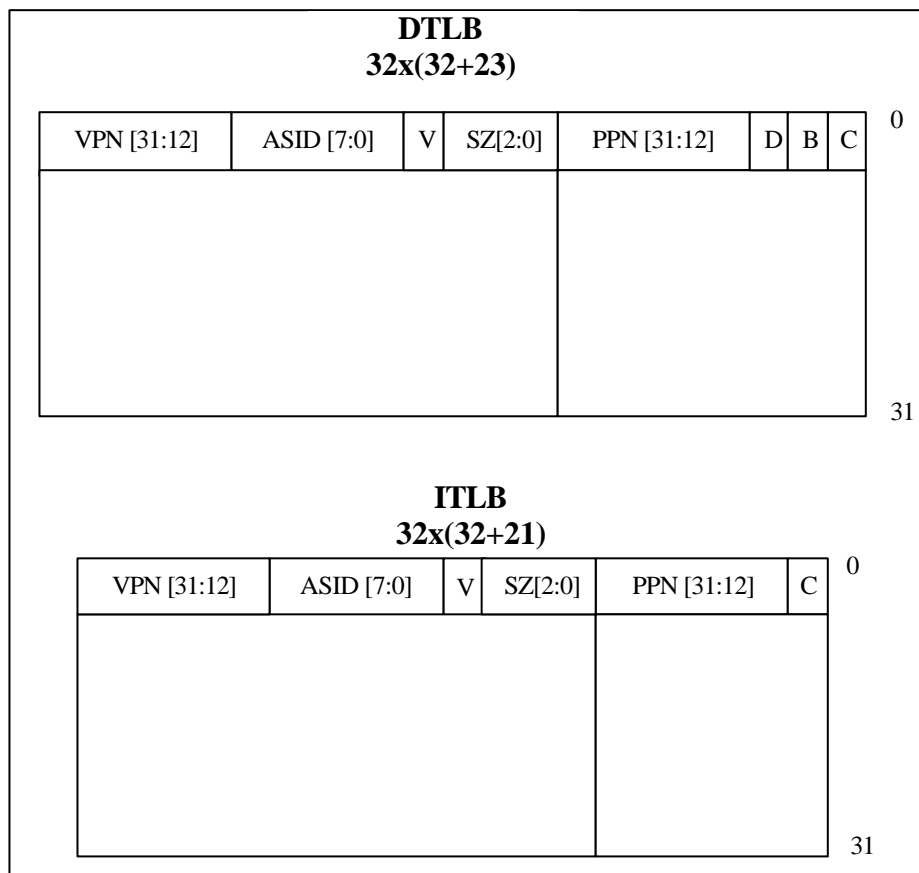


Figure 5-3 Configuration of TLB

- **VPN:** Virtual page number
 4Kbyte page: VPN bits [31:12] are valid.
 16Kbyte page: VPN bits [31:14] are valid.
 1Mbyte page: VPN bits [31:20] are valid
 16Mbyte page: VPN bits [31:24] are valid
- **ASID:** Address space identifier.
- **V:** Validity bit.
 Indicates whether the entry is valid or not.
 0: Invalid
 1: Valid
- **SZ[2:0]:** page size bits.
 Indicates the page size of this page. Only following 4 patterns are valid, other patterns will cause ITLB/DTLB search result unpredictable.
000: 4KB page size
001: 16KB page size

011: 1MB page size
111: 16MB page size

- **PPN:** Physical page number

Upper 20 bits of physical address

4Kbyte page: PPN bits [31:12] are valid physical page number.

16Kbyte page: PPN bits [31:14] are valid physical page number.

1Mbyte page: PPN bits [31:20] are valid physical page number.

16Mbyte page: PPN bits [31:24] are valid physical page number.

- **D:** dirty bit

Indicates whether a write has been performed to the page.

0: written has not been performed

1: written has been performed

- **B:** bufferable bit

Indicates whether a page is bufferable or not.

0: not bufferable.

1: bufferable

- **C:** cacheable bit

Indicates whether a page is cacheable, effective for mapped virtual memory space.

0: not cacheable

1: cacheable

Arca2 CPU core support four page sizes at the same time, if a big page overlaps with a little page, then the translation result of ITLB/DTLB is unpredictable.

Both ITLB and DTLB use round-robin replacement method. When write a new entry to ITLB/DTLB, overwrite the entry specified by round-robin pointer, and round-robin pointer add 1 to next line. If the pointer is 31 now, then the next is 0 in no ITLB/DTLB locked case.

Arca2 CPU core support ITLB/DTLB lock instruction to lock critical ITLB/DTLB entries, the locked entries are not overwritten by other virtual address nor invalidated by ITLB/DTLB invalidate instruction. Hardware will ignore the lock command if software is trying to lock the last entry of TLB, i.e. entry 31 can never be locked. When this happens, the entry will still be written into the ITLB/DTLB but the lock will be ignored.

Locked entries in ITLB/DTLB can be discarded by ITLB/DTLB discard configure operation, but the entries cannot be overwritten until all ITLB/DTLB is unlocked. If a write ITLB/DTLB instruction hit a locked entry, then the new physical address and page attributes will be written to the hit entry.

5.5 Address Translation Method

Figure 5-4 and
Figure 5-5 show flowcharts of memory accesses using the DTLB and ITLB respective.

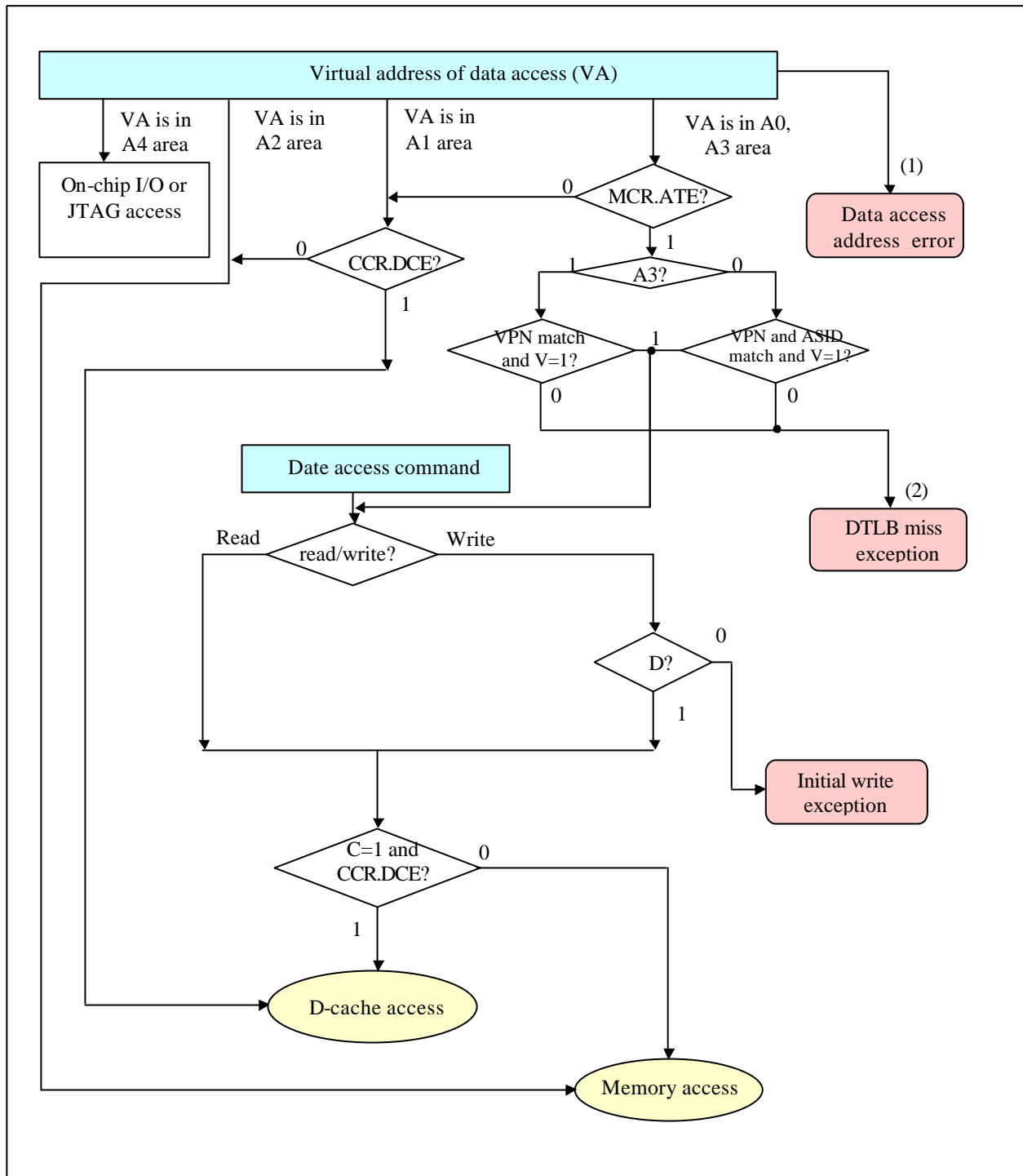


Figure 5-4 Flowchart of Data Access Using DTLB

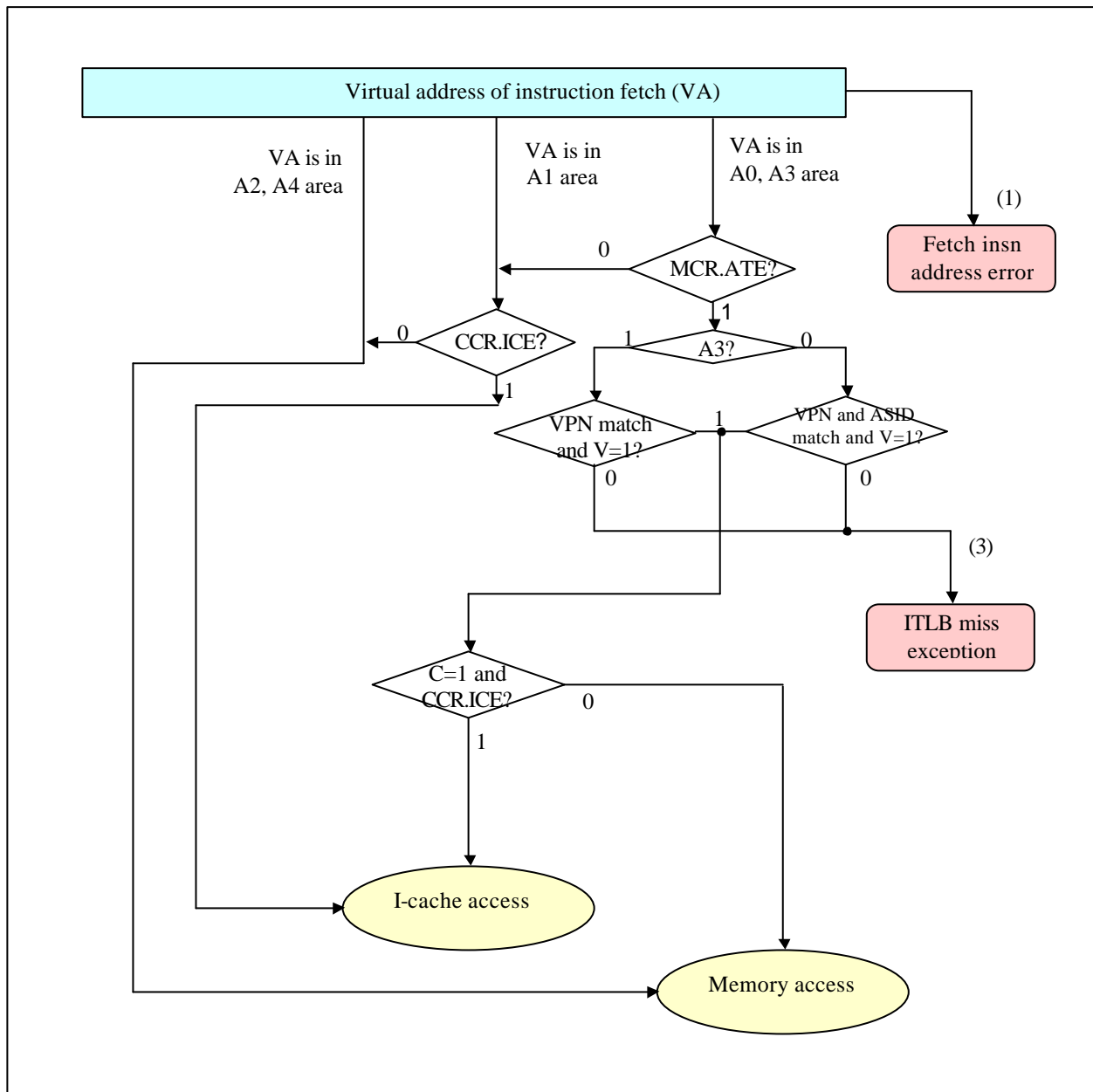


Figure 5-5 Flowchart of Instruction Fetch Using ITLB

Note:

- (1) When instruction or data access, MMU always check address error first (refer to 5.7.2) and send address error exception if address error is detected.
- (2) Since Arca2 CPU core uses virtual address D-cache and read access needn't page write protection check, so we will ignore DTLB miss exception when read access hit D-cache.
- (3) Since Arca2 CPU core uses virtual address I-cache and fetch instruction needn't page write protection check, we will ignore ITLB miss when fetch hit I-cache.

5.6 Configure Operation

5.6.1 MMU Function

Arca2 CPU core provides configuration interface between MMU and CPU to control MMU operation as listed below. All these instructions can be used in privileged mode only. ITLB instruction must be placed in unmapped area and at least 4 instructions following it cannot jump to mapped area. The first instruction following DTLB instruction cannot access mapped area for the consideration of that it may be depend on the result of these instructions.

- Discard TLB: clear V bit of hit entry in ITLB/DTLB.
- Invalidate TLB: clear all unlocked V bits of ITLB/DTLB.
- Read/Write TLB: read or write ITLB/DTLB.
- Lock TLB: write ITLB/DTLB and lock it if the address missed and there are at least two unlocked entries.
- Unlock TLB: unlocked all locked entries in ITLB/DTLB.
- CST/CLD: Read or write MMU control registers.

5.6.2 MMU Interface Format

The following table lists the detailed format and function of MMU configuration instructions.

Table 5-2 MMU configuration instruction

Operation	Code format	Function description
Discard ITLB	ITLB #discard, Rb	Search virtual address Rb in ITLB. If hit, clear the hit entry as invalid, else do nothing.
Invalidate ITLB	ITLB #inv, Rb	Invalid all unlocked ITLB entries, Rb is ignored.
Read ITLB	ITLB #read, Rb	Read ITLB data to CED: Search virtual address Rb in ITLB, if hit, clear 0 to CED.M bit, write the read data (PPN, SZ, C) to CED field respective, and CED.D,B bits undefined, else, set 1 to CED.M, and other bits undefined.
Write ITLB	ITLB #write, Rb	Store data reside in CED register to ITLB. Search virtual address Rb in ITLB, if hit, fill the full item of ITLB: VPN[31:12] from Rb, ASID from ASI register, PPN[31:12], SZ, C from CED register, and set 1 to V bit to hit entry, else fill them to replaced entry.
Lock ITLB	ITLB #lock, Rb	Search virtual address Rb in ITLB, if hit, do nothing; if miss, fill the full item of ITLB: VPN[31:12] from Rb, ASID from ASI register, PPN[31:12], SZ, C from CED register, and set 1 to V bit to replaced entry, and lock the entry.

Operation	Code format	Function description
Unlock ITLB	ITLB #unlock, Rb	Unlock all locked entries in ITLB. Rb is ignored.
Discard DTLB	DTLB #discard, Rb	Search virtual address Rb in DTLB. If hit, clear the hit entry as invalid, else, do nothing.
Invalidate DTLB	DTLB #inv, Rb	Invalid all unlocked DTLB entries. Rb is ignored.
Read DTLB	DTLB #read, Rb	Read data to CED: search virtual address Rb in DTLB, if hit read the physical information (PPN,SZ, D,B,C) to register CED, and clear 0 to CED.M bit, else set 1 to CED.M bit, other bits is undefined.
Write DTLB	DTLB #write, Rb	Store data reside in CED register to DTLB: search virtual address Rb in DTLB, if hit, write VPN (Rb[31:12]), ASID (register ASI), PPN, SZ, D, B, C (register CED), set 1 to V bit to the hit entry, else write to replaced entry in DTLB.
Lock DTLB	DTLB #lock, Rb	Search virtual address Rb in DTLB, if hit, do nothing; if miss, write VPN (Rb[31:12]), ASID (register ASI) valid, PPN, SZ, D, B, C (register CED), set 1 to V bit to the replaced entry in DTLB, and lock the entry.
Unlock DTLB	DTLB #unlock, Rb	Unlocked all locked entries in DTLB. Rb is ignored.
MMU Configure register load	CLD Ra, #MMU, #CR	Read data from MMU registers to Ra. CR holds control register's index number (refer to Table 5-1).
MMU Configure register store	CST Ra, #MMU, #CR	Write data of Ra to MMU registers. CR holds control register's index number (refer to Table 5-1).

Notes:

1. If the address is unmapped in “Discard”, “Read”, “Write”, and “lock” TLB instruction or MCR.ATE is zero, nothing is done.
2. If MCR.ATE is zero, “invalidate” instruction does nothing.

5.6.3 Code Examples

1. Fill or update a TLB entry is implemented by 2 instructions:

```

CST Ra, #MMU, #CED ! Record source data (PPN and attributes)
                   ! into CED register in MMU.
DTLB #write, Rb    ! Read data from CED register and write

```

! to DTLB entry specified by Rb.

2. Read an entry of TLB to check is implemented by 2 instructions:

```
DTLB #read, Rb      ! Read data from DTLB line specified by Rb,  
                    ! and write into CED register.  
CLD Ra, #MMU, #CED ! read data from CED register.
```

3. lock an entry to DTLB. In order to lock success, discard the entry or invalidate DTLB first.

```
CST Ra, #MMU, #CED ! Record source data (PPN and attributes)  
                    ! into CED register in MMU.  
DTLB #discard, Rb  ! discard the exist entry.  
DTLB #lock, Rb     ! write CED register and virtual page number  
                    ! to DTLB and lock this entry.
```

5.7 MMU exception

There are two kinds of exception in MMU: Data access Fault (D-fault) and Instruction fetch Fault (I-fault). The causes for D-fault and I-fault are explained in below:

Data access Fault may be induced by the following causes with priority from high to low:

- Illegal configure operation
- Data address error
- Data TLB miss
- Data TLB initial page write

Instruction fetch Fault may be induced by the following causes with priority from high to low:

- Instruction address error
- Instruction TLB miss

The priority of D-fault causes is higher than that of I-fault causes. When D-fault and I-fault occur at the same cycle, only the spot information corresponding to the exception cause with the highest priority will be recorded:

- Store the fault trigger address in MEA.
- Put the fault cause code into MCR.cause.

The D-fault and I-fault handlers share a common entry in the exception vector table. The exception routine needs to consult MCR.cause to further determine the exception service for the specific fault cause. There is a one bit (bit 31) in MCR.cause used to distinguish TLB miss or not so that the routine may use only two instructions to jump to the service handling TLB miss:

CLD Ra, #MMU, #MCR	; read control register number 0 in DTLB, i.e., MCR
BLT Ra, R0, label	; Ra < 0 means bit 31 of MCR == 1, i.e., TLB miss

The descriptions in below give further explanation for the fault causes.

5.7.1 Illegal configure exception

MMU checks the operation of configure instructions. If the operation is illegal, generates an illegal configure exception. Following configure cases are illegal:

- Use privileged CLD/CST command on MMU module in user mode.
- Use CLD/CST command on unknown module.
- Use CLD/CST command on unknown control register of MMU module.
- Use unknown command of ITLB, DTLB, I-cache, and D-cache.
- Use ITLB, DTLB instruction in user mode.
- Use I-cache, D-cache invalid, lock and unlock command in user mode.

5.7.2 Address Error

An address error for a fetch or data access occurs in the following cases:

- Instruction fetch address not located on word boundary.
- Instruction fetch address is other than A0 area in user mode
- Instruction (load, store, and swap) access address is misalign, that is, accessing a word/half-word not located on word/half-word boundary.

- Instruction (load, store, swap, I-cache discard, prefetch instruction, all D-cache user used instructions) access memory space other than A0 area in user mode.

5.7.3 TLB Miss

An ITLB miss occurs when the translation information of instruction fetch address cannot be found in the ITLB. A DTLB miss occurs when the translation information of data access address cannot be found in the DTLB.

Except normal load, store, and swap accessing, following D-cache instructions may occur DTLB miss too, if the address translation information cannot be found in DTLB:

- D-cache prefetch (only when D-cache miss first)
- D-cache allocation (only when D-cache miss first)
- D-cache p-lock (only when D-cache miss first)
- D-cache a-lock (only when D-cache miss first)

Except normal instruction fetch, I-cache prefetch and p-lock instruction will occur ITLB miss too, if the address missed in I-cache and translation information cannot be found in ITLB.

5.7.4 Initial Page Write

Initial page write is an exception of store (swap) access only. An initial page write for a store (swap) access occurs when, even though a DTLB entry contains the required address translation information, but the page attribute D is 0, which means no write has been performed to the current page.

6 Cache

6.1 Overview

Arca2 CPU core supports a Harvard structural one level cache (a respective instruction cache and data cache). Each has 8K bytes capacitance, and the cache line size is 8-word (32 bytes). Each line of the data cache has two dirty bits to specify the dirty situation of the datum in the upper and lower half line respectively. This document delineates the detail common specifications and especial features of the cache embedded in Arca2 CPU core. Please note that in Arca2 technical documents, data cache is abbreviated to D-cache, and instruction cache is abbreviated to I-cache.

To further reduce the memory access latency, Arca2 CPU core provides a 4-line X 4word/line write buffer.

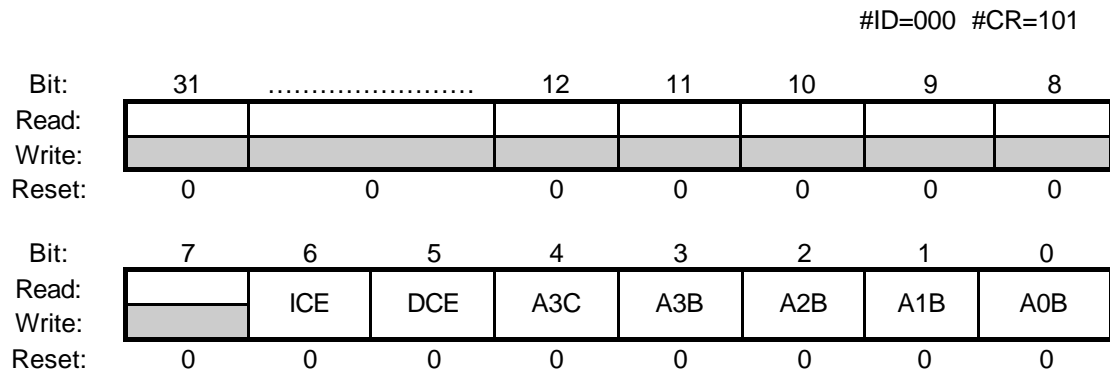
6.1.1 Cache Feature

Table 6-1 Cache Feature

Item	Features
Capacity	8 kilo bytes for I-cache and 8 kilo bytes for D-cache
Structure	32-way set associative
Line Size	32 bytes per line
Sets	8 sets
Write policy	Programmable WB (write back) and WT (write through) in D-cache
Write buffer	4 lines, 4 word per line in D-cache.
Hit algorithm	Virtual tag comparison
Replace method	Round-robin
Lock	Support lock and unlock function

6.2 Register Configuration

There is one control register named CCR to control the functional option of cache. Software can use CLD/CST instruction to access the register. The configuration of CCR shows in following figure.



Bit 31 ~ 7: Reserved bits, ignored in write operation, always 0 in read operation.

- **Bit 0 A0B:** Bufferable bit for A0 areas. 1: bufferable 0: unbufferable.
- **Bit 1 A1B:** Bufferable bit for A1 area. 1: bufferable 0: unbufferable.
- **Bit 2 A2B:** Bufferable bit for A2 area. 1: bufferable 0: unbufferable.
- **Bit 3 A3B:** Bufferable bit for A3 area. 1: bufferable 0: unbufferable.
- **Bit 4 A3C:** Cacheable bit for A3 area in direct map mode. 0: Uncacheable, 1: Cacheable
- **Bit 5 DCE:** D-cache enable. 0: Disable D-cache, 1: Enable D-cache
- **Bit 6 ICE:** I-cache enable. 0: Disable I-cache, 1: Enable I-cache

6.3 Data Cache and Write Buffer

Arca2 CPU core has one virtual tagged D-cache and one write buffer tightly combined with D-cache. Manipulation of D-cache and write buffer are controlled by the C (cacheable) and B (bufferable) fields of page table item resided in the DTLB, or by CCR.DCE and CCR.AxB, bits for direct map mode (MMU.ATE=0). Please refer to the MMU specification for address space partition definition, page table attribute specification and the CCR (Cache Control Register) contents description.

Combination cases of C and B attributes of page table item are listed in the below:

Table 6-2 Cacheable and Bufferable attribute of one page

MCR.ATE	Attribute	Virtual address area				
		A1	A2	A3	A0	A4
1	Cacheable	CCR.DCE	0	CCR.DCE & PT.C ^{*1}	CCR.DCE & PT.C ^{*1}	0
	Bufferable	CCR.A1B	CCR.A2B	PT.B ^{*1}	PT.B ^{*1}	0
0	Cacheable	CCR.DCE	0	CCR.DCE & CCR.A3C	CCR.DCE	0
	Bufferable	CCR.A1B	CCR.A2B	CCR.A3B	CCR.A0B	0

Note: *1 PT.C and PT.B represents the C and B attributes of page table item.

In above table, value 1 represents active state, while value 0 represents inactive state.

Table 6-3 describes D-cache and write buffer manipulation policy for cacheable and bufferable attribute of accessed address. From cache viewpoint, page attribute is invisible, cache only cares the attribute of the accessed address.

Table 6-3 D-cache and Write Buffer Policy

Cacheable	Bufferable	Policy
0	0	Non-cached, and non-buffered
0	1	Non-cached, and write data are buffered
1	0	Cached in write through(WT) policy, the write datum is buffered
1	1	Cached in write back(WB) policy, the replaced data are buffered

Notes:

- when A3C, A3B, A1B, or A0B of CCR are modified, all instructions that are located in the affected areas should be DISCARDED, and all data that are located in the affected areas should be DISCARDED or FLUSHed from DCACHE.
 - When PPN, C-bit of an ITLB entry is changed, all instructions that are located in the page should be DISCARDED.
 - When PPN, C-bit, D-bit or B-bit of an DTLB entry is changed, all data that are located in the page should be DISCARDED or FLUSHed.
- Otherwise, unpredictable results may occur.

6.3.1 D-cache Structure

The data cache is a 8-Kbyte, 32-way set associative cache. Following figure shows the structure of D-cache:

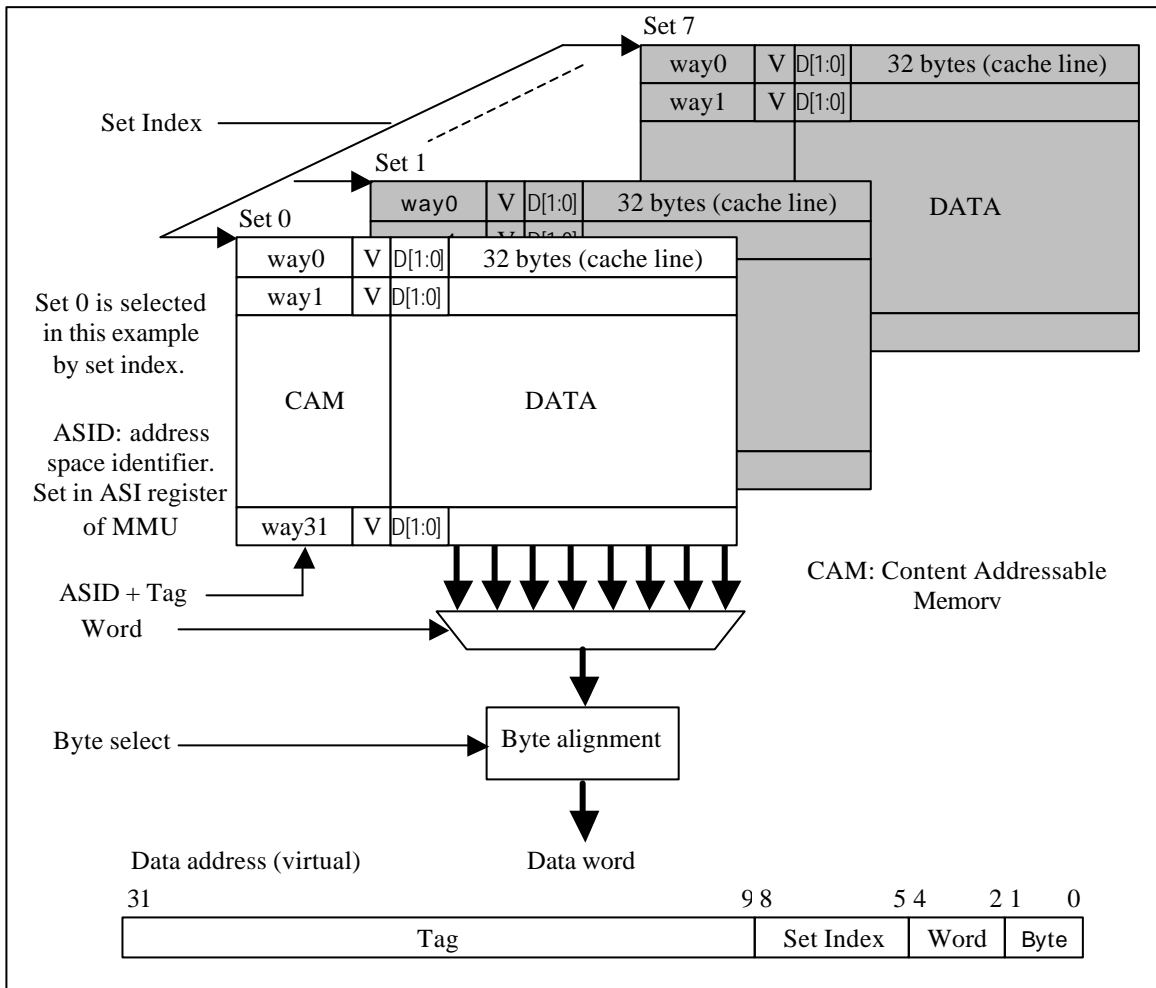


Figure 6-1 Data Cache Structure

The Figure 6-1 shows that there are 8 sets with each set containing 32 ways. Each way of a set contains 32 bytes (one cache line) and one valid bit. There also exist two dirty bits for every line, one for the lower 16 bytes and the other one for the upper 16 bytes. When a store hits the cache in write-back mode, one of the dirty bits associated with the hit line is set. The replacement policy is a round-robin algorithm.

Figure 6-1 shows the cache organization and how the data address is used to access the cache.

6.3.2 Cacheable Access Operation

After power-on reset, D-cache is disabled and all lines are invalid, it cannot be used until system initializes it. Set value 1 to CCR.DCE can enable Dcache. Please note that the write buffer cannot be disabled and serves only for bufferable or cacheable address areas. Refer to Table 6-2 and Table 6-3 for detail definition about address B (bufferable) and C (cacheable) attributes.

In later chapters, acronyms NB, NC, B and C are used to represent non-bufferable, non-cacheable, bufferable and cacheable. When D-cache is enabled, access cacheable memory may hit or miss D-cache. Following sections describe D-cache access hit/miss operation. Please note that Arca2 CPU core D-cache cache-line fill operation needs an 8-word (32 continuous bytes) wraparound burst read, while the cache write-back operation needs a 4-word (16 continuous bytes) wraparound burst write.

6.3.2.1 Read Access

Read Hit: When read access hits D-cache, the hit datum is transported from D-cache to CPU.

Read Miss: When reading D-cache misses, the write buffer should be searched first.

If the missed physical address (ignore the lowest 5 bits) hits the write buffer, the expected cache-line fill operation has to wait until the write buffer drains the hit line. Then the missed line aligned at 8-word boundary that contains the missed datum can be burst read from external memory and filled into D-cache.

As a contrast, failed search in the write buffer incurs an external memory burst read access immediately.

Moreover, if the replaced line calculated by round-robin algorithm contains dirty data, the dirty half line or all line will be backup to the write buffer, which is called WBB (write back to buffer). After D-cache completes the fill process, the final write back operation mandated by the write buffer due to preceding WBB (if exist) will be manipulated implicitly.

Please note that the leadoff datum (word granularity) returned from external memory must be the one that triggers the miss operation and expected by CPU, and the other data filling in the background of CPU if no memory access or core configure instruction during the duration

6.3.2.2 Write Access

Arca2 CPU core data cache supplies the **write without allocate** strategy for WT access, and **write with allocate** strategy for WB access. Write operation needs at least two cycles (hit D-cache or miss in WT mode), but if no memory access or core configure instruction following it, the write finished in backend to CPU.

Write Hit (WB policy): the hit datum is only written to D-cache, no external memory access request is asserted. Value 1 should be set to the correlative dirty bit that represents the dirty situation of the correlative half line (upper or lower according to the accessed address).

Write Hit (WT policy): the hit datum is written to D-cache, meanwhile it is written to the write buffer, and the datum buffered in the write buffer will be written to external memory later. In the case, the dirty bit is cleaned.

Write Miss (WB policy): When a write miss occurs, the write buffer should be searched first. If the missed address (ignore the lowest 5 bits) hits the write buffer, the cache-line fill operation has to wait until the write buffer drains the hit line. Then the missed line aligned at 8-word boundary that contains the missed datum can be burst read from external memory and filled into D-cache. As a contrast, failed search in the write buffer incurs an immediately external memory burst read access. Moreover, if the replaced line calculated by round-robin algorithm contains dirty data, the dirty half line or all line will be backup to the write buffer. Similar to read miss, a WBB (if exist) causes the write buffer to perform a write back operation implicitly. Please note that the missed datum (word granularity) should be returned from external memory first, and is stuffed to D-cache after being coalesced with the new write datum issued by CPU. Value 1 should be set to the correlative upper or lower dirty bit field of the filled line according to the accessed address.

Write Miss (WT policy): When a write miss occurs in WT mode, the missed datum is just stored to the write buffer, and is written to the external memory later by the write buffer.

6.3.3 Non-cacheable Access Operation

When D-cache is disabled, access operation does not look up Dcache. However, in the case that D-cache is enabled and the accessed address is non-cacheable, software must ensure that the address does not exist in the D-cache. Otherwise, the access result is unpredictable.

6.3.3.1 Read Access

When the write buffer is empty, assert a single beat read request to the external memory immediately, otherwise, assert the read request after write buffer drains.

6.3.3.2 Write Access

When the accessed address is B, the write datum is stored to the write buffer without directly asserting single beat write request to the external memory. When the accessed address is NB, the write buffer should be scanned first. If the scanning result shows that the write buffer is not empty, it has to wait until the write buffer drain that the write request can be asserted. Otherwise, the write request is immediately asserted.

Write operation needs at least two cycles (bufferable and write buffer is not full), but if no memory access or core configure instruction following it, the write finished in backend to CPU.

6.3.4 Write Buffer

Arca2 CPU core has one four-level FIFO type write buffer, it always attempts to send buffered data to the external memory as long as the external bus is idle until it evicts all the buffered data. The write buffer can overlap some memory access latency with CPU core's pipeline operation. However, the performance enhancement derived from the write buffer tightly depends on the code optimization and the cache hit rate. In general, interspersing store instructions in the code stream is better than issuing such instructions in bulk. For cacheable memory access, higher cache hit rate (both L-cache and D-cache) is helpful to overlap writing external memory latency. Arca2 CPU core write buffer supports following operations:

- Buffer the replaced half lines that contain dirty data (these data should be written back to the external memory) until the external bus is usable.
- Buffer the write datum until the external bus is usable if the address is with WT policy or with NC and B attribute.
- Empty itself to ensure that all the buffered data arrive to the external memory before a new access request issued by CPU, which is called flushing write buffer
- An uncacheable read/swap or uncacheable & unbufferable write will empty the write buffer before access external memory.

6.4 Instruction Cache

Arca2 CPU core has one virtual tagged and physical indexed I-cache. Following figure shows the configuration of I-cache. The instruction cache is a 8-Kbyte, 32-way set associative cache. Each way of a set contains 32 bytes (one cache line) and one valid bit. The replacement policy is round-robin algorithm.

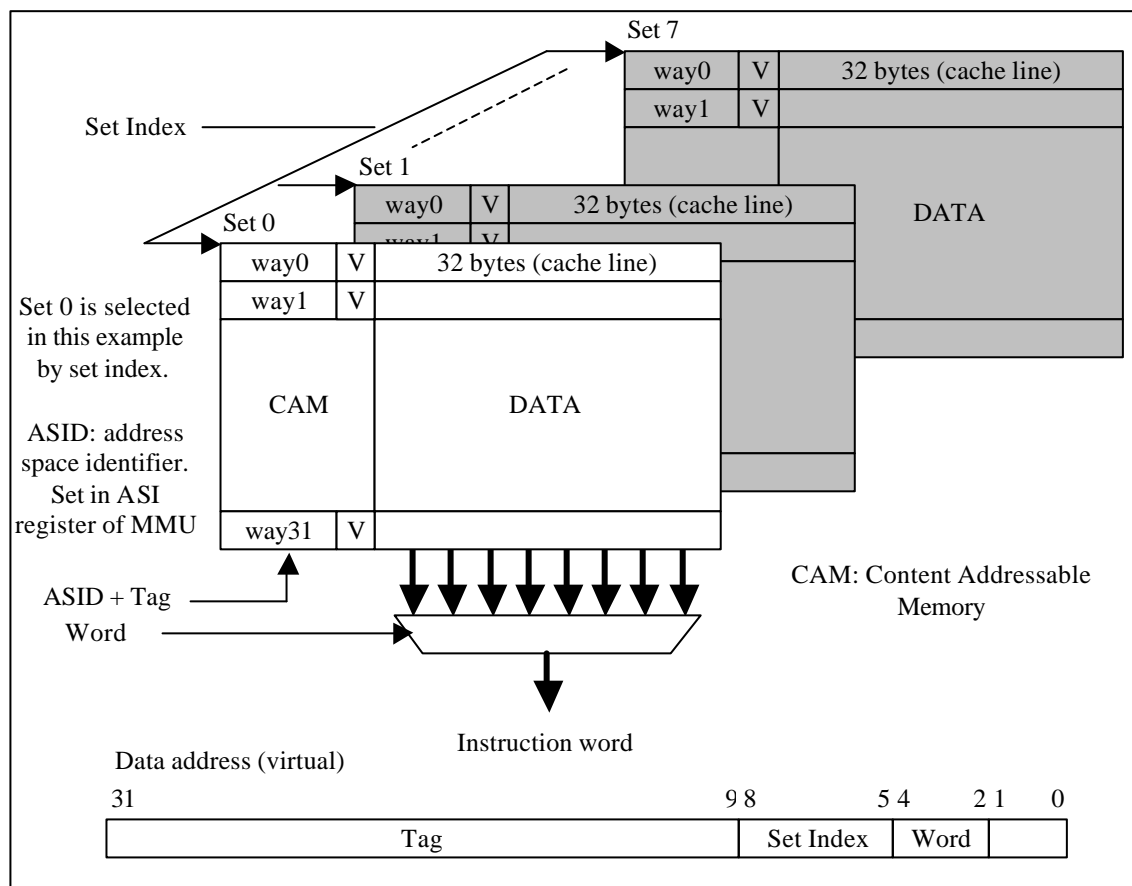


Figure 6-2 Instruction Cache Structure

6.4.1 Fetch Operation

When I-cache is enabled (CCR.ICE = 1) and the fetch address is cacheable, every requested instruction is searched in the I-cache.

Fetch hit: A fetch hit occurs when the requested instruction is found in the I-cache, the hit instruction is returned to CPU.

Fetch miss: A fetch miss occurs when the requested instruction is not found in the I-cache. When this occurs, I-cache sends a fetch request to the external memory, and 8 words aligned at 32-byte boundary would be burst read back, like D-cache fill operation, the missed instruction should be returned first. The following fetch may hit the filling data or hit I-cache under the last filling process.

When I-cache is disabled (CCR.ICE = 0) or the fetch address is non-cacheable, each fetch request need fetch one instruction (word granularity) from the external memory.

6.5 Prefetch Operation

Arca2 CPU core supports prefetching data into D-cache. It is a backend action, that is, once CPU successfully issues a prefetch instruction, pipeline should go on without waiting prefetched data being filled into D-cache. Please note that any data access fault caused by prefetch must be handled like the normal load type instruction. During prefetching period, any new memory access instruction or core configure instruction will have to wait until prefetching finish. So the sophisticated Interspersing of the prefetch instructions can reduce D-cache miss rate. Because the prefetched data are in the cacheable memory and will be accessed by CPU soon. In essence, such manipulation is used to get high cache hit rate and overlap external memory access latency.

Arca2 CPU core supports prefetching instruction into I-cache too. It is a backend action, and following fetch can hit I-cache or hit the filling data. But I-cache and D-cache access the same external memory, so we need be care for avoid data access and instruction fetch access external memory at the same time for the higher performance.

6.6 SWAP Operation

When CPU issues a swap instruction, which results in a read operation followed by a write operation. And the two operations are in-dividable, in another word, they are combined into an atomic operation. According to the combinatorial case of C and B attribute of the swapped address, following actions maybe performed:

- C&B: The operation is similar to a write access with WB policy, except the original datum in the swapped address (may be in the D-cache or in the memory) should be returned to CPU before being updated by the write operation.
- C&NB: When the read operation hits D-cache, the hit datum is returned to CPU from D-cache, then D-cache and write buffer is updated by the write operation. When the read operation misses D-cache, after the write buffer drains, the coupled read/write operation is asserted to external bus, and the correlative bus controller must ensure that it is an atomic operation, note that no fill operation here and D-cache is not updated in this case.
- NC&B: The coupled read/write operation is asserted to external bus after the write buffer drains, and the correlative bus controller must ensure that it is an atomic operation.
- NC&NB: The coupled read/write operation is asserted to external bus after the write buffer drains, and the correlative bus controller must ensure that it is an atomic operation.

6.7 Alias Solution

Alias occurs when multiple virtual addresses map to a same physical address. When cache is virtual tagged or virtual indexed, such aliases are illegal because without taking any special correcting method, updating one alias cache line will not be snooped by other associative alias cache lines. Hence, data coherency problem may arise when accessing other untouched alias lines later.

Arca2 CPU core automatically solves the alias problem by hardware, in detail, for D-cache, it ensures that only one copy of contents in a physical memory region exists in the D-cache at any time for associative alias lines. But if the virtual memory system is in direct mapping mode, the alias of A0 area for different ASID isn't solved. And D-cache DISCARD, FLUSH, WRITE_BACK commands don't check alias hit, because any alias hit command will clean (write back dirty data) the hit line.

For I-cache, maybe more than one copy of contents in a physical memory region exists, fortunately, as the I-cache is read only, no data coherency problem delineated before may be triggered. However, memory coherency problem still exists among I-cache, D-cache, write buffer and external memory. Thus, software has to make careful management.

6.8 Coherency Between Cache And External Memory

Software must ensure data coherency between the cache and the memory. For the cacheable memory shared by Arca2 CPU core and another device, the latest modified data may be recorded in D-cache, or in the shared memory.

- To guarantee the data coherency when the latest modified data resided in D-cache, flush the crucial D-cache lines, if some lines are dirty, write back operations are generated then. Otherwise, these lines are simply abandoned. Flush write buffer is necessary before the shared crucial data can be accessed from memory.
- To guarantee the data coherency when the latest modified data resided in the shared cacheable memory, just invalidate the associated lines in the I-cache or D-cache.

Moreover, coherency problem between I-cache and D-cache can also be solved by above two methods.

6.9 Cache replacement and lock function

Arca2 CPU core Icache and Dcache use round-robin replacement method. Every cache set has a replace pointer. When cache need allocate a new line, the new address will overwrite the line specified by the replace pointer, and replace pointer add 1 to specify the next line. If current pointer is 31, then next is 0 in no locked line case.

Instructions and data can be locked in Icache and D-cache so that they cannot be overwritten by linefill. This operates with a granularity of one line, that is 8 words (32 bytes).

Software has the ability to lock performance critical routines into the instruction cache. Up to 31 lines in each set can be locked. Hardware will ignore the last lock command if software is trying to lock all lines in a particular set, i.e. way 31 can never be locked.

When use ICACHE/DCACHE lock instruction to lock one address to CACHE, it must be sure that the address must miss in ICACHE/DCACHE. If the address of lock command had existed in ICACHE/DCACHE, the result of ICACHE/DCACHE lock instruction is unpredictable. So before lock ICACHE/DCACHE, it is better to use ICACHE/DCACHE discard or invalidate instruction to clear the address from ICACHE/DCACHE.

The locked line can be discard, write back, and flush, but the hole in locked area after doing discard and flush cannot be filled with new line unless unlock operation is done. ICACHE/DCACHE invalidate instruction will not clear the locked line.

Please note that DCACHE discard, write back, and flush instruction do not check alias hit, but P-lock and A-lock instruction need check alias hit, and the result is unpredictable if P-lock or A-lock instruction alias hit DCACHE. When read, write, swap, prefetch, and allocate instruction alias hit a locked line, it will change the locked virtual tag in DCACHE and write back dirty data of the line.

ICACHE/DCACHE unlock instruction used to unlock all the locked lines in ICACHE/DCACHE.

6.10 Cache Configuration

Cache configure instructions is used to serve the OS development or application program optimization. Such operations use the instruction format: **ICACHE CMD, Rb, S10** and **DCACHE CMD, Rb, S10**. Please refer to Arca2 ISA for detail information.

6.10.1 Operation List

Table 6-4 list all the operations to D-cache and I-cache through the core module interface of Arca instruction set.

Table 6-4 Cache Operations

Operation	Code	Function description
Discard one I-cache line	ICACHE #discard, Rb, S10	Discard specific line in the I-cache. An expected instruction virtual address ($Rb + S10 \ll 2$) is sent to I-cache, if it hits I-cache, the hit line is invalidated, otherwise, nothing is done.
Invalidate I-cache	ICACHE #inv, Rb, S10	Invalidate all unlocked I-cache lines, Rb and S10 are ignored. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a protection fault.
Prefetch I-cache	ICACHE #prefetch, Rb, S10	Fetch specified line to I-cache. Search virtual address ($Rb + S10 \ll 2$) in I-cache, if hit, do nothing, else allocate a new line for fill it from external memory.
Prefetch and lock one I-cache line	ICACHE #p-lock, Rb, S10	Fetch specific line to I-cache and lock it. Search virtual address ($Rb + S10 \ll 2$) in I-cache, if hit, do nothing, else allocate a new line for fill it from external memory and locked it. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a protection fault.
Unlock all locked line in I-cache	ICACHE #unlock, Rb, S10	Unlock all locked lines in I-cache. These lines will be overwritten when I-cache miss later. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a protection fault.
Prefetch one data line	DCACHE #pfd, Rb, S10	If specific cacheable virtual address ($Rb + S10 \ll 2$) hits D-cache, nothing is done; if the address alias hit D-cache, write back dirty data of the hit line and mark the line with clean label; otherwise, allocate one line for the address and load the associated data from external memory.
Discard one D-cache line	DCACHE #discard, Rb, S10	If specific cacheable virtual address ($Rb + S10 \ll 2$) hits D-cache, the hit line is invalidated; otherwise, nothing is done. Please be cautious to use the function because it does not write back any dirty data.
Invalidate D-cache	DCACHE #inv, Rb, S10	Invalidate all unlocked D-cache lines, Rb and S10 are ignored. Be cautious to use the function because it does not write back valid dirty lines. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a

		protection fault.
Flush one D-cache line	DCACHE #flush, Rb, S10	If specific cacheable virtual address ($Rb + S10 \ll 2$) hits D-cache, invalidate the hit line, moreover, if the line is dirty, the dirty upper or/and lower half lines should be written back to external memory. If the address miss D-cache, nothing is done.
Write back one dirty line	DCACHE #wb, Rb, S10	If specific cacheable virtual address ($Rb + S10 \ll 2$) hits D-cache and the hit line is dirty, then write back the dirty upper or/and lower half lines, then set 0 to the dirty bit field of the hit line. If the address misses D-cache or the hit line is not dirty, nothing is done.
Allocate one D-cache Line	DCACHE #alloc, Rb, S10	If specific virtual address ($Rb + S10 \ll 2$) hits D-cache, nothing is done. If the address alias hits D-cache, write back dirty data of the line and mark the line with clean label. Otherwise, allocate one line for the address directly and do not load associated data from external memory. Moreover, if the allocated line calculated by round-robin algorithm has valid dirty data, they will be written back to external memory before allocation. Please note that after allocation, some valid data must be written to the line before read, otherwise, the read datum is a random value.
Allocate and lock one D-cache line	DCACHE #a-lock, Rb, S10	Allocate a new line in D-cache and lock it. Search virtual address ($Rb + S10 \ll 2$) in D-cache, if hit, do nothing; if the address alias hit D-cache, write back dirty data of the hit line and mark the line with clean label; otherwise allocate one line for the address directly and needn't load associated data from external memory then lock it. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a protection fault. Moreover, if the allocated line calculated by replace algorithm has valid dirty data, they will be written back to external memory before allocation. Please note that after allocation, some valid data must be written to the line before read, otherwise, the read datum is a random value.
Prefetch and lock one D-cache line	DCACHE #p-lock, Rb, S10	Fetch specified line to D-cache and lock it. Search virtual address ($Rb + S10 \ll 2$) in D-cache, if hit, do nothing; if the address alias hit D-cache, write back dirty data of the hit line and mark the line with clean label; otherwise allocate a new line for fill it from external memory and lock it. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a protection fault. Moreover, if the allocated line calculated by replace algorithm has valid dirty data, they will be written back to external memory before fill.
Unlock all locked lines in D-cache	DCACHE #unlock, Rb, S10	Unlock all locked lines in D-cache. These lines will be replaced when D-cache miss later. This operation is only permitted in supervisor mode, in user mode, issuing the instruction will trigger a

		protection fault.
Flush write buffer	DCACHE #flush-buf, Rb, S10	Evict all buffered data in the write buffer to external memory until it is empty, Rb and S10 are ignored.

Notes:

- Except invalidat, p-lock, a-lock, unlock I-cache/D-cache, issuing other cache instructions in user mode is valid. Moreover, address protection mechanism (refer to MMU exception section for detail) is also suitable for those cache instructions.
- Prefetch, allocate, p-lock, and a-lock operation only can be performed when the expected address has cacheable attribute.
- Discard, flush and write back operation issued by specific cache instructions do not care the cacheable attribute of the expected address. Hence, it had better flush the correlative D-cache lines of a page before adjust its C attribute in the page table, otherwise, some unpredictable result may be triggered.

6.10.2 Code Examples

6.10.2.1 Discard one D-cache line

If some dirty data in D-cache are useless later, they can be discarded directly. Such operation does not trigger any write back action, which can reduce bus traffic and enhance the program executing efficiency in some special applications.

```
DCACHE #discard, R3, 0      ! R3 contains the expected address
```

6.10.2.2 Invalidate D-cache

```
DCACHE #inv, R0, 0          ! Invalidate all unlocked D-cache lines.
                             ! After the instruction executing, only
                             ! the locked address can hit the D-cache.
                             ! The function can be used, for instance,
                             ! to ensure data coherency of shared
                             ! memory with WT attribute
```

6.10.2.3 Flush one D-cache line

Flush dirty line in D-cache to enforce the dirty data to be written back to external memory and filled from external memory in next access.

```
DCACHE #flush, R3, 0        ! R3 contains the expected address, flush
                             ! valid dirty line can incur write back
                             ! operation, which enforce the memory
                             ! section associated with the flushed
                             ! dirty line to be updated by dirty data
```

6.10.2.4 Write back dirty line

Following code sequence attempts to write back a valid dirty line to external memory, and make it as a valid clean line again.

```
DCACHE #wb, R3, 0          ! R3 contains the expected address
```

6.10.2.5 Allocate D-cache line

This command can be used in following two cases:

- Allocate a line in the D-cache, which associative address will be accessed by store type instruction first, and the original contents in the address are not concerned at all. For instance, when stack is cacheable, use the function to allocate some new stack space in the D-cache before push some never cached data, which can reduce stack push load evidently. Be care for the granularity of allocating is one line of the D-cache, which are 8 words (32 bytes).
- Evict all dirty data in D-cache back to external memory during context switching. In Arca2 CPU core, following codes can clean all unlocked data in D-cache: (It is better to use an unmapped address as s_addr. If the lowest 11 bits are all zero, two ORI instructions in following codes can be removed. Because the branch instruction will lost a cycle in Arca2 CPU core, you can unroll the loop by using DCACHE #alloc, R1, S10 to improve the performance.)

```
LHI    R1, (s_addr>>11)
ORI    R1, R1, (s_addr & 0x7ff)
LHI    R2, ((s_addr+0x2000)>>11)
ORI    R2, R2, ((s_addr+0x2000) & 0x7ff)
ALP:
DCACHE #alloc, R1, 0
ADDI   R1, R1, 32
BNE    R1, R2, ALP
DCACHE #inv, R0, 0
```

6.10.2.6 Flush write buffer

Consider a code runtime modification application, before branching to the dynamically generated code section, it had better flush D-cache and write buffer to synchronize the memory contents, because the fresh generated code maybe still buffered in the write buffer. And then discard the address in I-cache to force it fill from external memory.

```
DCACHE #flush, R4, 0
ICACHE #discard, R5, 0
DCACHE #flush-buf, R6, 0
JA     R1, R2, 0          ! R2 contains branch target
                          ! R1 contains returned address
```

6.10.2.7 Discard I-cache line

If a segment of instruction has been modified in main memory, we need update the new instruction to I-cache, i.e. we should discard the old copy of the segment in I-cache. The following code sequence discards a segment of instruction code in I-cache:

```

LOOP:
ICACHE #discard, R3, 0    ! R3 contains the address of invalidated
                          ! code
ADDI   R3, R3, #32        ! let R3 point to next line
BLE    R3, R4, #LOOP      ! R4 contains the end address

```

6.10.2.8 Invalidate I-cache

If there are more instructions have been modified in main memory, we can use ICACHE #invalidate instruction to discard all unlocked copy in I-cache to improve the performance of discard.

```

ICACHE #inv, R0, 0        ! Invalidate all unlocked I-cache lines.
                          ! After the instruction executing,
                          ! only locked address can hit I-cache.

```

6.10.2.9 Lock cache

We can use cache lock instruction to lock critical instruction or data into I-cache or D-cache to improve the performance. Following codes give an example to lock a segment critical instruction to I-cache.

```

LHI    R3, (s_addr >> 11)
ORI    R3, (s_addr & 0x7ff) ! load start address of critical
                          ! code to R3

LHI    R4, (e_addr >> 11)
ORI    R4, (e_addr & 0x7ff) ! load end address of critical code
                          ! to R4

ICACHE #inv, R0, 0        ! invalidate I-cache to make sure
                          ! the critical instruction address
                          ! will miss in I-cache first.

LOOP:
ICACHE #lock, R3, 0
ADDI   R3, R3, #32        ! let R3 point to next line
BLE    R3, R4, #LOOP

```

7 Debug and JTAG

7.1 Overview

Generally software debugger should modify program memory in order to insert breakpoints. The breakpoint can be either a real breakpoint instruction or an illegal instruction, which can trigger break or illegal-instruction exception. When this kind of instructions is encountered the debugged program is stopped and the debugger accepts the control right from it. Additional external hardware tools can supplement these basic mechanisms, such as logic analyzers and in-circuit emulators (ICEs) for additional control and information about program execution.

Although this model of debug works well for many sorts of system, it has the following shortcomings if the system to be debugged is a highly-integrated design:

- System-On-a-Chip (SOC) component design no longer provides an external interface to the processor pin-out or system bus, making the use of logic analyzers and ICEs difficult or impossible.
- Debugging based on the insertion of software breakpoint instructions assumes that programs reside in RAM. It is impractical for fully ROM-based systems and assuming support in the O/S for these techniques.
- For consumer electronic applications, a communication port like Ethernet or RS-232 serves no purpose beyond software debug and adds disproportionately to the cost and size of the design.
- The ROM necessary to support a debug monitor on a consumer electronic application could add unacceptable costs.

The Debug module with extended JTAG supplements the ARCA architecture in dealing with these problems. The processor can be tied into a JTAG scan chain and comprehensively debugged using an external JTAG probe connected to the system's JTAG TAP interface.

The Debug module offers efficient and smart break functions to simplify program debugging. It provides the following new capabilities for software and system debug:

- Off-board JTAG memory
The extended JTAG allows an ARCA processor in host-monitoring debug to refer to instructions or data that are not resident on the system under test. This JTAG memory is mapped to the processor as if it were physical memory; references to it are converted into transactions on the TAP interface. Both instructions and data can be accessed in JTAG memory, which allows debugging of systems without requiring the presence of a ROM monitor or debugger scratchpad RAM. It also provides a communications channel between debug software executing on the processor and an external debugging agent.
- Support hardware breakpoints
 1. instruction fetch breakpoint
 2. data access address/result breakpoint
 3. asynchronous break/bootThese breakpoints can be used to implement watchpoints, breakpoints and single-step execution, without requiring that the program code reside in RAM.
- System access via the extended JTAG interface

The extended JTAG can force processor entry into host-monitoring debug. Debug software can then get further system access via JTAG interface.

7.1.1 Debug Features

Table 7-1 Debug Features

Break condition	Description
Fetch address match	Compare instruction address bus with preset value
Fetch address mask	Ignore masked bits when fetch address comparison
Access address match	Compare data address bus with preset value
Access address mask	Ignore masked bits when data access address comparison
Address ASID match	Compare ASID of current process with preset value
Access data match	Compare data access (store/swap) result with preset value
Access type control	Only specific access type (read, write, etc.) can be monitored
Access size control	Only specific access size (byte, half word, word) can be monitored
Software breakpoint	Provide a software-breakpoint instruction SBRK
Asynchronous break	Host send a command to break current executing instruction stream
Asynchronous boot	Host send a command to reboot processor from dedicated memory

7.1.2 Extended JTAG Feature

Table 7-2 Extended JTAG Features

Break condition	Description
IEEE 1149.1 standard	Compliant with standard JTAG feature
Extended data register	BIU_BSR is composed of boundary cells encompassing the interface of BIU in ARCA CPU core
Extended instructions	In addition to standard mandatory instructions: BYPASS, extended JTAG in ARCA supports seven new instructions named ASYN_BRK, ASYN_BOOT, CONTROL, ADDR, DATA, ALL and HOST_MODE

7.1.3 Debugging Pattern

- **Self-Monitor Debugging**
In this debug pattern, both the debugged program and the debugger itself are executing on the ARCA processor. Since JTAG interface cannot be accessed in the case, asynchronous break and asynchronous boot cannot be triggered.
- **Host-Monitor Debugging**
In this debug pattern, the debugged program is executing on the ARCA processor, while the debugger is executing on the host. After debug handler begins executions, the debugger can communicate with and even control the ARCA processor through JTAG interface.

7.1.4 Debug & JTAG Solution Diagram

Debug & JTAG solution apparently shows that in some situation, CPU core can communicate with external world through standard TAP ports instead of the core bus agent. Therefore, host-monitor can be realized through this communication method. However, when TAP does not work, the Debug module can still monitors IU_BUS to support self-monitor through the software debugger's help.

The TAP in the CPU core (called internal TAP in later chapters) only supports extended seven new instructions and standard BYPASS instruction, while the TAP outside of the CPU core (called external TAP in later chapters) just supports standard JTAG instructions including BYPASS, SAMPLE/PRELOAD, EXTEST and RUNBIST. The selection of internal TAP and external TAP depends on the pin TAP_SEL when power-on reset. If TAP_SEL is 1, internal TAP works, otherwise external TAP works. This document has no reference about detail of external TAP since it is an IEEE 1149.1a standard compatible product.

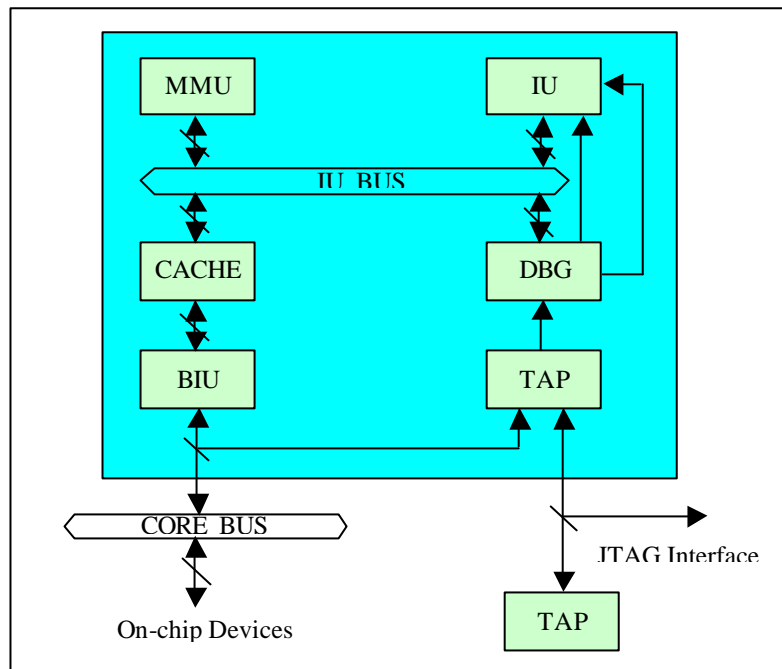


Figure 7-1 Debug and Extended JTAG

7.2 Extended JTAG

7.2.1 Overview

ARCA series JTAG hardware are compliant with IEEE 1149.1 TAP&BSA standard. In addition, to support host-monitoring debug, several private instructions and an extra boundary cell data register is added. The following diagram shows the structure of the internal TAP.

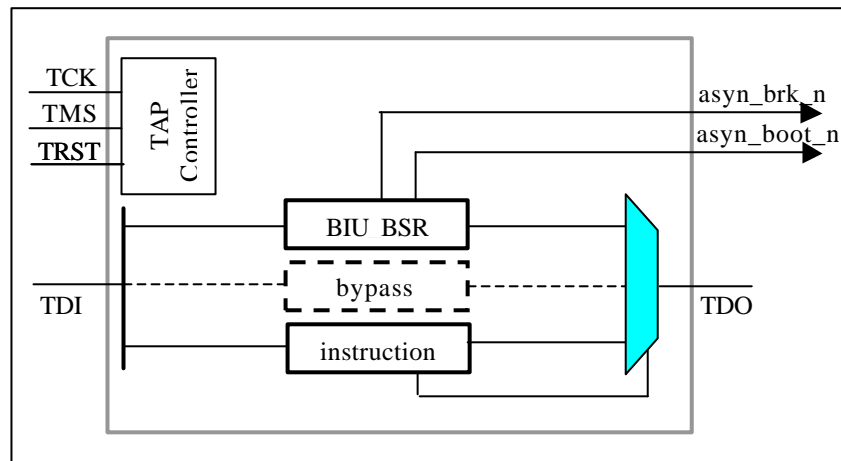


Figure 7-2 Internal Tap

Note:

The extra output signals from BIU_BSR register are devised to serve for host-monitoring. Later chapters will discuss them in detail.

7.2.2 Standard & Extended Private Instructions

There are seven private instructions appended to support the communication between ARCA processor and host. The following table lists the instructions supported by internal TAP.

Table 7-3 Extended JTAG Instructions

Name	Code	Description
BYPASS	H'F	Compliant with JTAG standard, but the output from TDO is not the value of TDI latched at the rising edge of TCK and it is a random value.
CONTROL	H'0	After decoding, always select the sub-field of BIU_BSR named CTRL, which is devised for host polling convenience
ADDR	H'1	After decoding, always select the sub-field of BIU_BSR named ADDR
DATA	H'2	After decoding, always select the sub-field of BIU_BSR named DATA. Note that only the instruction can inject data from JTAG memory to CPU core at Update_DR state of internal TAP
ALL	H'3	After decoding, select the whole BIU_BSR between TDI and TDO
ASYN_BRK	H'4	After decoding, assert an active low level signal named asyn_brk_n, attempt to asynchronously break CPU core current operation. In addition, it selects the sub-field of BIU_BSR named CTRL between TDI and TDO
ASYN_BOOT	H'5	After decoding, assert an active low level signal named asyn_boot_n to boot processor from specific off-board JTAG memory space. In addition, it selects the sub-field of BIU_BSR named CTRL between TDI and TDO
HOST_MODE	H'6	After decoding, the exception vector table base (VBR) is automatically re-directed (not modified) and fixed at H'EC000000. Which is called host-mode vector base. The instruction always selects the sub-field of BIU_BSR named CTRL between TDI and TDO. Note that only active TRST_ or at least 5 high level TMS cause the internal TAP controller to enter into the Test-Logic-Reset state, which can restore the vector base to initial value.
	H'7~E	Reserved for future usage

Note: Any one of the seven private instructions is active, which denotes that the JTAG memory space can be accessed.

7.2.3 Extended Data Registers

In the diagram of TAP&BSA, there is 1 data register. Its usage is listed below.

- BIU_BSR represents the boundary scan registers encompassing the CPU core, which is used to realize communication between JTAG interface and CPU core.

In the host-monitoring environment, the contents of the BIU_BSR are cyclically shifted out or updated according to the monitoring result. The following figure shows the format of BIU_BSR.

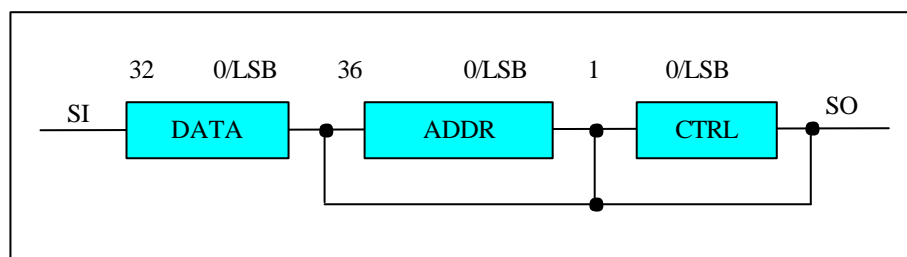


Figure 7-3 BIU_BSR Register

Since there are two forwarding paths after DATA and ADDR, the three sub-fields of BIU_BSR can be manipulated respectively. The following table lists these sub-fields' configuration.

Table 7-4 BIU_BSR Register

Name	Bit field	Description
CTRL	bit0	Active value 1 denotes an active JTAG memory space access request, inactive value 0 means no JTAG memory access
	bit1	Read/Write label of access request. 0: read access, 1: write access.
ADDR	bit31~0	Denotes the access address
	bit33~32	Denotes the access result size. 2'B11: byte, 2'B01: half word, 2'B10: word.
	bit34	Bus lock label. Active 1 means current read access and next write access are an atomic operation. The bit can be ignored by host because not any other bus device connects the bus serving for internal TAP.
	bit35	Active 1 denotes burst enable. Inactive 0 denotes single beat access.
	bit36	Active 1 denotes 8 words burst; inactive 0 denotes 4 words burst. When bit35 takes inactive 0, the field is ignored.
DATA	bit31~0	Access result. For read access of JTAG memory space, need be filled by JTAG interface (through DATA instruction). For write access of JTAG memory space, need be scanned out by JTAG interface.
	bit32	IU_BUS freezing label. 1: bus is ready, 0: bus is locked due to an unfinished access. For an active JTAG memory space access, need JTAG interface (through DATA instruction) set active high level to unfreeze the IU_BUS

7.2.4 Endian Adjustment

Since ARCA series support big-endian and little-endian, debugger in the host maybe has to adjust endian if JTAG memory use different endian. The following figure shows the byte position in the DATA part according to access size and the two LSBs of ADDR part.

SIZE	ADDR[1:0]	Big Endian	Little Endian								
byte	2'B00	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0
	3	2	1	0							
	3	2	1	0							
	2'B01	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0
3	2	1	0								
3	2	1	0								
2'B10	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	
3	2	1	0								
3	2	1	0								
2'B11	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	
3	2	1	0								
3	2	1	0								
half word	2'B00	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0
	3	2	1	0							
3	2	1	0								
2'B10	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	
3	2	1	0								
3	2	1	0								
word		<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0	<table><tr><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	3	2	1	0
3	2	1	0								
3	2	1	0								
NOTES: gray area represents available byte field											

Figure 7-4 Endian Adjustment

7.2.5 JTAG Memory Space

JTAG memory is dedicated to serve for debugger in the host-monitor environment. The memory space from H'EC000000 to H'FFFFFFF is mapped to off-board JTAG memory space. The space can not be allocated to any other external device. The following table lists the JTAG memory space special usage.

Table 7-5 JTAG Memory Space

Memory Space	Usage
H'EC000000	Special vector serving for DBG_BOOT
H'EC000000~H'EC00001F	Position of vector table for host-mode vector base . In the case, reset handler entry is always equivalent to boot handler entry.
H'FFFFFF00 ~ H'FFFFFFF	Memory mapped JTAG control register space. H'FFFFFF20 : Asynchronous break response register, when IU writes any value to the register, which denotes that the asynchronous break has already been responded by IU. The sustained active signal asyn_brk_n then is deasserted by hardware automatically

7.2.6 Miscellaneous Constraints

To make host-debug environment safe and reliable, some hardware constraints are imposed. System designers need pay more attention to this section.

7.2.6.1 Reset Constraint

In ARCA series, TRST_ is necessary. During power-up reset (due to pressing power key), TRST_ must be active to reset the internal TAP. However, during hot reset (system reset, without pressing power key), TRST_ is not affected at all.

7.2.6.2 TCK Constraint

To guarantee the processor can enter boot handler through JTAG interface instead of reset handler in the case that, memory system has some problem or even there is not any on-board memory system, the following rules should be obeyed:

- Core clock frequency is not less than TCK clock frequency
- Maximum TCK clock frequency is 50MHz

7.2.6.3 TMS Constraint

Since there are two TAPs on the chip, which need a special pin to control the TMS usage. For instance, when the pin TAP_SEL keeps at high level, TMS is switched to serve for internal TAP, and the path of TMS routing to the external TAP is locked at high level. When the pin TAP_SEL keeps at low level, the handling is reversed.

7.3 Debug Module

The Debug module supports instruction and data access hardware breakpoints. By monitoring the IU_BUS, ASID of current process (for multi-process OS), and combining with the presetting control information, the Debug module can trigger abundant types of hardware breakpoints, such as single step breakpoint, precise watchpoint, ambiguous address range breakpoint and etc. The following figure shows the topology of Debug in the CPU core.

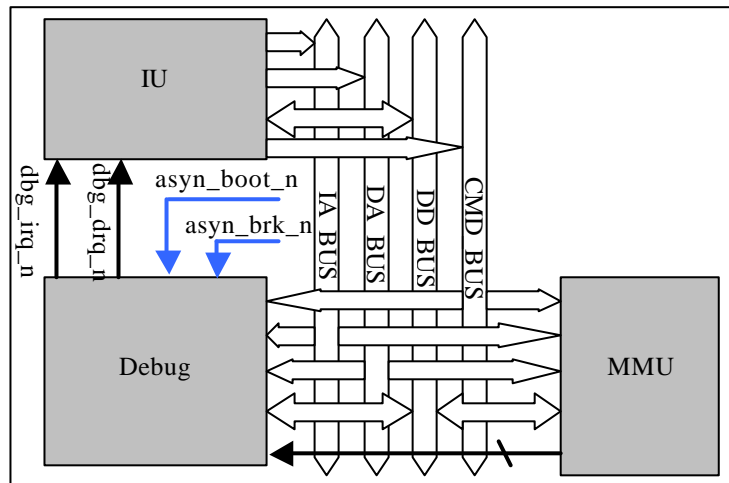


Figure 7-5 The Stutcture of Debug Module

Table 7-6 Debug Internal Signals

Signal Name	Description
IA_BUS	IU instruction fetching address bus
DA_BUS	IU data access address bus
DD_BUS	IU data bus
CMD_BUS	IU access&fetch command bus
dbg_irq_n	break request signal, due to fetch breakpoint, asynchronous break
dbg_drq_n	break request signal, due to data access breakpoint
asyn_boot_n	signal from BIU_BSR to boot processor from JTAG memory space
asyn_brk_n	signal from BIU_BSR to break current executing instruction stream
asid	ASID of current executing process

7.4 Debug Register Configuration

The following table lists the Debug registers. Software can use CLD/CST instruction to access the register. These registers can only be read/written in supervisor mode.

Table 7-7 Debug Registers

Name	Full Name	R/W	Initial value when power on	Access Size	# ID	#CR
DBG_CR	Debug Configure register	R/W	H'00000000	32	011	000
DBG_IA0	Debug Instruction Address 0	R/W	Undefined	32	011	001
DBG_IA1	Debug Instruction Address 1	R/W	Undefined	32	011	010
DBG_DA0	Debug Data access Address 0	R/W	Undefined	32	011	011
DBG_DA1	Debug Data access Address 1	R/W	Undefined	32	011	100
DBG_DD0	Debug Data access (store/swap) Result 0	R/W	Undefined	32	011	101
DBG_ASID	Debug ASID	R/W	Undefined	32	011	110

7.4.1 Register Descriptions

7.4.1.1 Debug Control Register(DBG_CR)

#ID=011 #CR=000

Bit:	31	30	29	28	27	26	25	24
Read:	HOST	BKEN						
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	23	22	21	20	19	18	17	16
Read:						SBRK	DHIT	
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	15	14	13	12	11	10	9	8
Read:	IHIT		DMEN	IMEN	ASIDM	DMSK0	DLEN1	
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	7	6	5	4	3	2	1	0
Read:	DLEN0		DRW1		DRW0		DDEN	IDEN
Write:								
Reset:	0	0	0	0	0	0	0	0

Bit 29 ~ 19: Reserved bits, ignored in write operation, always 0 in read operation.

- IDEN: Hardware instruction fetch breakpoint enable.
0: disable instruction fetch breakpoint.
1: enable instruction fetch breakpoint.

- DDEN: Hardware data access breakpoint enable.
 - 0: disable data access breakpoint.
 - 1: enable data access breakpoint.
- DRW0: Data access type label for DBG_DA0.
 - 2'H0: match due to read or write memory access can be granted.
 - 2'H1: match due to read access can be granted.
 - 2'H2: match due to write access can be granted.
 - 2'H3: reserved.
- DRW1: Similar to DRW0, serve for DBG_DA1.
- DLEN0: Data access size label for DBG_DA0.
 - 2'H0: word access is checked.
 - 2'H1: half word access is checked.
 - 2'H2: byte, half word and word access are all checked.
 - 2'H3: byte access is checked.
- DLEN1: Similar to DLEN0, serve for DBG_DA1.
- DMSK0: Data access (store/swap) result mask for DBG_DD0.
 - 0: denotes that data result is not cared for data access breakpoint monitoring.
 - 1: denotes that data result is cared for data access breakpoint monitoring.

Note that only DBG_DA0 support data access result breakpoint.
- ASIDM: ASID mask enable.
 - 1: denotes that current ASID is not cared for any breakpoint monitoring (fetch or data access).
 - 0: denotes that current ASID must be monitored for hardware breakpoint.
- IMEN: Fetch breakpoint mask enable.
 - 0: denotes that DBG_IA1 is used for DBG_DI0 mask bits. In DBG_IA1, 0: Corresponding address bit is compared, 1: Corresponding address bit is masked.
 - 1: denotes that DBG_IA1 is also used as the second fetch address register, hence the two fetch address registers must be set with precise expected values to perform non-maskable address comparing.
- DMEN: Data access breakpoint mask enable.
 - 0: denotes that DBG_DA1 is used for DBG_DA0 mask bits. In DBG_DA1, 0: Corresponding address bit is compared, 1: Corresponding address bit is masked.
 - 1: denotes that DBG_DA1 is also used as the second data access address register, hence the two data access address registers must be set with precise expected values to perform non-maskable address comparing.
- IHIT: Fetch breakpoint match flag.
 - The field is automatically set with value 1 by hardware once such a matching is detected.
 - Bit14 denotes DBG_IA0 match, and bit15 denotes DBG_IA1 match.
- DHIT: Data access breakpoint match flag.
 - The field is automatically set with value 1 by hardware once such a matching is detected.
 - Bit16 denotes DBG_DA0 match, and bit17 denotes DBG_DA1 match.
- SBRK: Software breakpoint request flag.
 - This field is automatically set when a software-breakpoint instruction (SBRK) is executed.
- BKEN: Asynchronous break request flag.

If the instruction in internal TAP is ASYN_BRK, then it is set to 1 and an exception is generated immediately. However it will be kept 1 until the instruction has been changed and it has been acknowledged through writing 0xEFFFFFF20.

0: denotes no asynchronous break request occurs.

1: denotes an active asynchronous break request being asserted.

The bit is read-only by software.

- HOST: Host-monitoring environment enable.

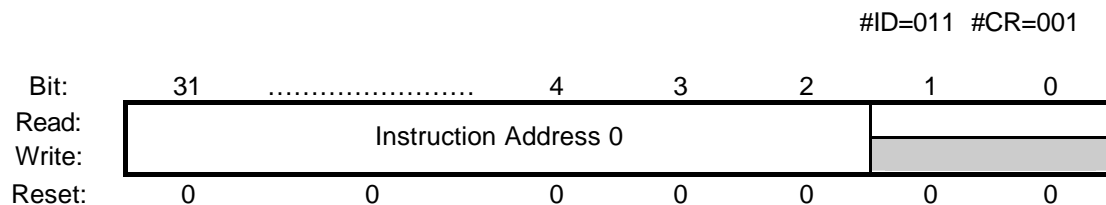
If the instruction in internal TAP is CONTROL, ADDR, DATA, ALL, ASYN_BRK, ASYN_BOOT or HOST_MODE, then it is 1, otherwise it is 0.

0: denotes that accessing JTAG memory space **cannot** be granted by host through JTAG interface.

1: denotes that accessing JTAG memory space **can** be granted by host through JTAG interface.

The bit is read-only by software.

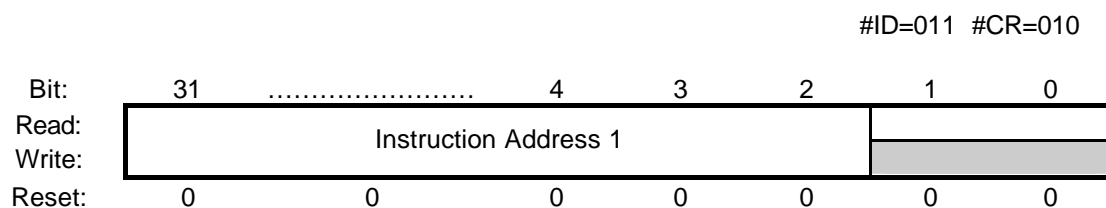
7.4.1.2 Instruction Address Register 0(DBG_IA0)



Bit 1 ~ 0: Reserved bits, ignored in write operation, always 0 in read operation.

Bit 31~2: Set expected fetch address in this register to trigger instruction fetch breakpoint.

7.4.1.3 Instruction Address Register 1(DBG_IA1)

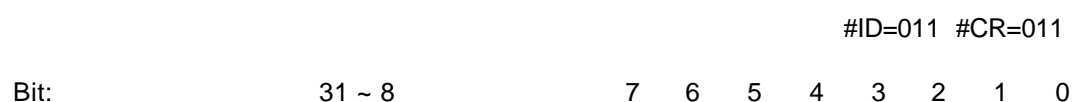


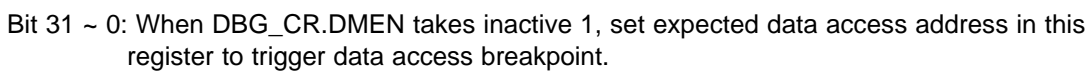
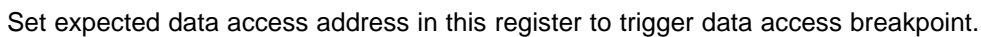
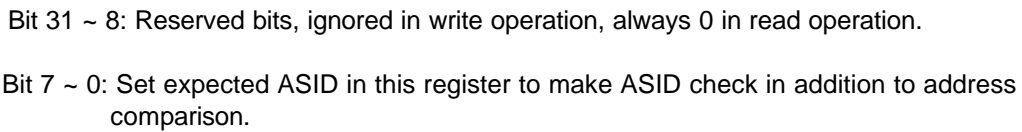
Bit 1 ~ 0: Reserved bits, ignored in write operation, always 0 in read operation.

Bit 31~2: When DBG_CR.IMEN takes inactive 1, set expected fetch address in this register to trigger instruction fetch breakpoint.

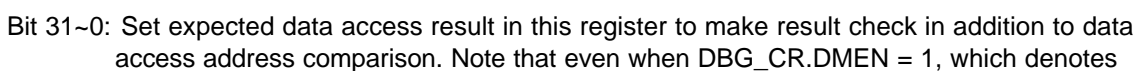
When DBG_CR.IMEN takes active 0, which can mask the comparing result of instruction address[31:2]. For example, set 32'HFFFFFFF in DBG_IA1, which may trigger a fetch breakpoint for each instruction fetching despite of fetch address once the DBG_CR.IDEN is set and SR.DE is set (refer to exception spec for SR usage).

7.4.1.4 Address Space Identifier Register(DBG_ASID)





Note that there has no initial value for this register, when DBG_CR.DMEN is inactive high, which serves as another data access address register.



that two access address should be monitored, data access result breakpoint can only be triggered for DBG_DA0. It only works for store and swap instructions.

7.5 Debug Operation

7.5.1 Overview

In general, the Debug module in ARCA supports two kinds of hardware breakpoint break condition: synchronous break and asynchronous break. When SR.DE is set active value 1, which denotes that IU permits break exception, any available asserted break request can be responded by IU.

For synchronous break, the Debug monitors the IA_BUS of IU for instruction fetch breakpoint condition matching detection, and monitors the DA_BUS and DD_BUS of IU for data access breakpoint matching detection. Once the captured value from monitored target match the expected break conditions that are set before, active low level signals dbg_irq_n and/or dbg_drq_n then are asserted to inform IU that some preset breakpoint is triggered.

The break caused by software breakpoint instruction (SBRK) is also a synchronous one. However this break is controlled by SR.DE.

1. If SR.DE = 1, SBRK cause a debug exception, DSR/DPC are used to save env, SR.DS is set to 1, handler is load from vector offset = 2 (the debug vector), exception return address is next instruction.
2. If SR.DE = 0, SBRK cause an illegal instruction exception, ESR/EPC are used to save env, SR.DS is set to 0, handler is load from vector offset = 1 (the illegal insn vector), exception return address is this instruction

Moreover, the Debug module supports asynchronous break mechanism including asynchronous break and asynchronous boot. Note that only in host-monitoring environment such breaks may be active. **IF SR.DE is disabled, the signal asyn_brk_n is ignored, while asyn_boot_n does not care for SR.DE.** For asynchronous break, once the signal asyn_brk_n takes active low level, the break request dbg_irq_n is immediately asserted to attempt to break current executing code stream, and the signal can retain at low level until the asyn_brk_n is de-asserted. For asynchronous boot, once the signal asyn_boot_n takes active low level and the pipeline is not frozen, the processor can switch to supervisor mode immediately and redirects execution to an asyn-boot handler routine.

Note that the entry of the asyn-boot handler routine is placed at JTAG memory space H'EC000000, independent with exception vector table.

7.5.2 Fetch Breakpoint Operation

Fetch breakpoint can be used to trace the flow of an executing code segment. To trigger a fetch breakpoint, the following steps are needed:

1. set DBG_CR.IDEN with inactive value 0, to avoid inadvertently fetch breakpoint triggering
2. set DBG_IA0, DBG_IA1, DBG_ASID with expected values
3. set DBG_CR.IMEN and DBG_CR.ASIDM with expected value, and enable breakpoint detection mechanism by setting DBG_CR.IDEN with active value 1

After finishing above steps, once an effective fetch matches the presetting break condition, and so long as SR.DE is set with active value 1, then dbg_irq_n can be asserted. Accompanied with asserting dbg_irq_n, DBG_CR.IHIT (IHIT[0] and/or IHIT[1]) is set active 1 to represent that a fetch breakpoint is captured. See the following true table for detail.

Table 7-8 Fetch Breakpoint True Table

DBG_CR.IDEN	SR.DE	DBG_IA (*1)	DBG_ASID (*2)	dbg_irq_n
0	--	--	--	inactive
1	0	--	--	inactive
1	1	mismatch	mismatch	inactive
		mismatch	match	inactive
		match	mismatch	inactive
		match	match	active

Notes:

1. Asterisk (*1) denotes that comparison of fetch address is masked by DBG_IA1 when DBG_CR.IMEN takes active 0. When DBG_CR.IMEN takes inactive 1, the match of DBG_IA means at least one of DBG_IAX (x represents 0, 1) is identical with the captured instruction address bus.
2. Asterisk (*2) denotes that comparison of ASID is ignored when DBG_CR.ASIDM takes inactive 1. However, when DBG_CR.ASIDM takes active 0, the match of DBG_ASID means that DBG_ASID is identical with current ASID.
3. Double dash line denotes don't care field.

7.5.3 Data Access Breakpoint Operation

Data access breakpoint can be used to monitor the memory access situation. To trigger a data access breakpoint, following 4 steps are needed:

1. set DBG_CR.DDEN with inactive value 0, avoid inadvertently fetch breakpoint triggering
2. set DBG_DA0, DBG_DA1, DBG_ASID, DBG_DD0 (optional) with expected values
3. set DBG_CR.DRW0, DBG_CR.DRW1, DBG_CR.DLEN0, DBG_CR.DLEN1, DBG_CR.ASIDM, DBG_CR.DMSK0 and DBG_CR.DMEN with expected values, and enable breakpoint detection by setting DBG_CR.DDEN with active value 1.

After finishing above steps, once an effective data access matches the presetting break condition, and so long as SR.DE is set with active value 1, then dbg_drq_n can be asserted. Accompanied with asserting dbg_drq_n, DBG_CR.DHIT (DHIT[0] and/or DHIT[1]) is set active 1 to represent that a data access breakpoint is captured, meanwhile DBG_DA0 is overwritten by captured DA_BUS when DBG_CR.DMEN is active 0. One point must be emphasized that comparison of DBG_DD0 depends on the set of DBG_CR.DMSK0. When DBG_CR.DMSK0 takes value 0, which represents data access address breakpoint, otherwise, represents data

access (store/swap) result breakpoint. The following true table lists the detailed match process of data access breakpoint. **Moreover, data access (store/swap) result breakpoint only serves for DBG_DA0, and only write access result case is supported.**

Table 7-9 Data Access Breakpoint True Table

DBG_CR .DDEN	SR .DE	DBG_DAx (*1) & DBG_ASID (*2)	DBG_CR .DMSK0	DBG_CR .DRWx	DBG_CR .DLENx (*3)	DBG_DD0	dbg_drq_n
0	--	--	--	--		--	inactive
1	0	--	--	--		--	inactive
1	1	mismatch	--	--		--	inactive
		match	==0	mismatch	mismatch	--	inactive
				mismatch	match	--	inactive
				match	mismatch	--	inactive
				match	match	--	active
			==1	mismatch	mismatch	--	inactive
				mismatch	match	--	inactive
				match	mismatch	--	inactive
				match	match	mismatch	inactive
				match	match	match	active

Notes:

1. Asterisk (*1) denotes that comparison of data access address is masked by DBG_DA1 when DBG_CR.DMEN takes active 0. When DBG_CR.DMEN takes inactive 1, the match of DBG_DA means at least one of DBG_DAx (x represents 0, 1) is identical with the captured data address bus.
2. Asterisk (*2) denotes that comparison of ASID is ignored when DBG_CR.ASIDM takes inactive 1. However, when DBG_CR.ASIDM takes active 0, the match of DBG_ASID means that DBG_ASID is identical with current ASID.
3. Asterisk (*3) contains a special case. That is, if DBG_CR.DLENx is set by value 2, the match of the LSB[1:0] of DBG_DAx must refer to current access length, simply put, it's a covering over data access breakpoint. For instance, set value 2 to DBG_CR.DLEN0, bring on following three cases:
 - 1) For current word size access, DBG_DA0[1:0] comparing is always match despite of comparing result described in (*1).
 - 2) For current half word size access, DBG_DA0[0] comparing is always match despite of comparing result described in (*1), while DBG_DA0[1] still refers to the comparing result carried out from (*1)
 - 3) For current byte size access, the match of DBG_DA0[1:0] always refers to the comparing result carried out from (*1).
4. Double dash line denotes don't care field.

7.5.3.1 Data Access Command Monitoring

In ARCA ISA, there are several special system instructions serving for system special usage. These instructions also need touch IU_BUS, but the Debug module does not monitor them at all.

Moreover, when IU issues a SWAP instruction, the Debug module regards it as a store type instruction for data access result breakpoint monitoring. The following table lists the instructions that are not monitored.

Table 7-10 Instructions That Are Not Monitored

Instruction Name	Notes
CLD/CST Rx, #ID, S10	Value on the DA_BUS is not concerned
ITLB #CMD, Rx	Value on the IA_BUS is not concerned
DTLB #CMD, Rx	Value on the DA_BUS is not concerned
ICACHE #CMD, Rx, S10	Value on the IA_BUS is not concerned
DCACHE #CMD, Rx, S10	Value on the DA_BUS is not concerned

7.5.4 Asynchronous Break/Boot Operation

ARCA Debug supports asynchronous break, and the break type is dedicated to host-monitor debugging. ARCA JTAG extends two instructions named ASYN_BRK and ASYN_BOOT. When instruction register of the internal TAP decodes ASYN_BRK, as a result, the signal asyn_brk_n is asserted to Debug; similarly, the signal asyn_boot_n is asserted to IU for ASYN_BOOT. Note that DBG_CR.DDEN and DBG_CR.IDEN just control the synchronous hardware breakpoint comparing logic, e.g., they do not forbid request of asynchronous break source. Once the Debug module samples the active low asyn_brk_n signal, it immediately asserts active dbg_irq_n to IU, and in the case, dbg_irq_n can sustain until the asyn_brk_n is deasserted.

Note that once the ASYN_BRK instruction is asserted, before the memory mapped register H'FFFFFF20 is written by a store type instruction, the signal asyn_brk_n can maintain at active status despite whether current JTAG instruction is ASYN_BRK. In order to make asyn_brk_n inactive, CTRL should be shifted into JTAG instruction and the memory mapped register H'FFFFFF20 should be written. Software should check DBG_CR.BKEN bit to judge whether asyn_brk_n has been inactive.

For ASYN_BOOT, there is no need to acknowledge it and asyn_boot_n is automatically deasserted.

7.6 Debug Exception Operation

When an effective `dbg_irq_n` or `dbg_drq_n` is asserted, IU should respond them immediately, unless some other higher priority exception requests arise simultaneously. For active low `asyn_boot_n` caused by `ASYN_BOOT`, which takes the highest priority over all of other exceptions even including reset and does not care for `SR.DE`. The following fragment of exception resource signals' priority table illustrates all debug exceptions priority cases.

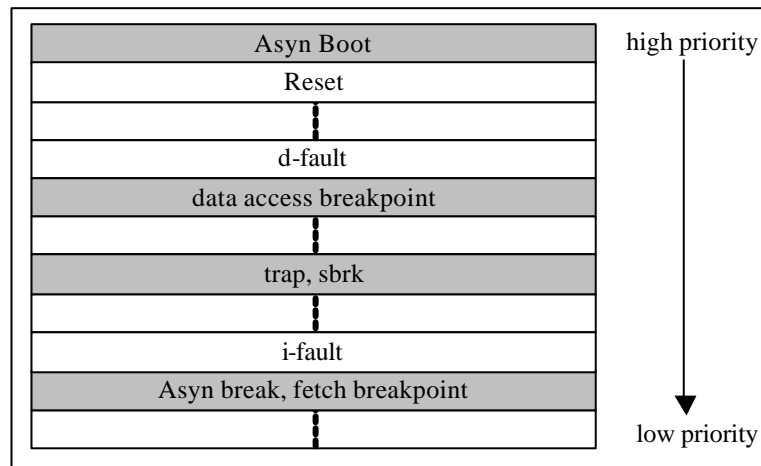


Figure 7-6 Exception Priority

Once a debug exception is granted by IU, the following common steps need be done automatically:

1. Load debug handler entry from exception vector table
2. Save expected return address to DPC
3. Save SR to DSR
4. Set value 0 to `SR.DE` to avoid reenter problem in debug handler
5. Set value 1 to `SR.SM` to let CPU toggle to supervisor mode
6. Set value 1 to `SR.DS` to represents an active debug state

One point must be emphasized that in the fetch breakpoint or data access breakpoint handler routine, it had better remove the `DBG_CR[IHIT]` or `DBG_CR[DHIT]` after investigate them by software, otherwise, the obsolete match flags may confuse later breakpoint event judgment, or cause even more worse situation: the same fetch breakpoint is triggered endlessly. In detail, if both `DBG_CR[IHIT]` and `DBG_CR[DHIT]` are set, just get rid of `DBG_CR[DHIT]` according to above exception priority diagram rather than remove them all. If only one of the two breakpoints is triggered, just remove `DBG_CR[IHIT]` or `DBG_CR[DHIT]`, in terms of the breakpoint type. However, in the handler routines of other exception types, do not attempt to affect those breakpoint flag bits, otherwise, some concurrently captured breakpoint events may miss out.

The later sections delineate the responding process for 6 types of debug break in detail. It is better to refer to IU exception spec for more information.

7.6.1 Fetch Breakpoint Debug Exception

Fetch breakpoint is similar to I-fault, it is also an IF stage exception, hence instructions in the later pipeline stages from ID to WB will be done normally. However, in some pipelining situations, special exception handling tactics (for instance, ID stage instruction is canceled too^{*1}) may be taken to manipulate the acknowledgement process of the fetch breakpoint. Even though, at any case, hardware can guarantee that after the handler routine of fetch breakpoint, returning location always points to the first canceled instruction in the code stream, or the executed instruction which resuming has not any side effect. As an instance, a fetch breakpoint next to a BCC instruction (BCC is untaken) need always return to the location of the BCC instruction, rather than itself, besides, hardware can manage the case to avoid endless loop. Please refer to exception model spec for more detail. When IU responds a fetch breakpoint, the following actions shall be done.

For hardware (necessary):

1. Load common debug handler entry from exception vector table (base + H'8)
2. Save returning location value to DPC
3. Save SR to DSR
4. Set value 0 to SR.DE to avoid reenter problem in debug handler
5. Set value 1 to SR.DS and SR.SM

For software (recommend, not necessary):

1. Read DBG_CR.IHIT, DBG_CR.DHIT and DBG_CR.SBRK to check that which breakpoint is granted
2. After enter the subroutine service for software breakpoint, clear DBG_CR.IHIT
3. When service finish, use RTE instruction to return to the normal execution stream

Note: *1 When fetch break occurs, if the previous instruction is a BCC, a MUL with Rh not R0 stalled for one cycle, or any instruction stalled for two cycles, then this previous instruction is cancelled and it is set to the return address DPC. In this situation, except for BCC case, the fetch break will occur again after return from handler and continue the execution.

7.6.2 Data Access Breakpoint Debug Exception

Data access breakpoint is similar to D-fault, it is also an MA stage exception source. However, differ from D-fault, the trigger instruction in the MA stage should be executed, that is, only instructions from IF to EX stage need be canceled. If IU grants a data access breakpoint, following actions may be done.

For hardware (necessary):

1. Load common debug handler entry from exception vector table (base + H'8)
2. Save return address that point to the current EX stage instruction to DPC
3. Save SR to DSR
4. Set value 0 to SR.DE to avoid reenter problem in debug handler
5. Set value 1 to SR.DS and SR.SM

For software (recommend, not necessary):

1. Read DBG_CR.IHIT, DBG_CR.DHIT and DBG_CR.SBRK to check that which breakpoint is granted
2. After enter the subroutine service for software breakpoint, clear DBG_CR.DHIT
3. When service finish, use RTE instruction to return to the normal execution stream

Evidently, if data access breakpoint handler does not change any of the data access break conditions, after handler executing, the next adjacent instruction can be naturally executed.

7.6.3 Software Breakpoint Debug Exception

If SR.DE == 0, SBRK causes an illegal instruction exception. Here we only consider the SBRK is used as a software breakpoint at SR.DE == 1.

Software breakpoint is similar to trap, it is also an ID stage exception, hence instructions in the later pipeline stages from EX to WB will be done normally. At any case, hardware can guarantee that after the handler routine of SBRK, the returning location is next instruction. When IU responds a software breakpoint, the following actions shall be done

For hardware (necessary):

1. Load common debug handler entry from exception vector table (base + H'8)
2. Save returning location value to DPC
3. Save SR to DSR
4. Set value 0 to SR.DE to avoid reenter problem in debug handler
5. Set value 1 to SR.DS and SR.SM

For software (recommend, not necessary):

1. Read DBG_CR.IHIT, DBG_CR.DHIT and DBG_CR.SBRK to check that which breakpoint is granted
2. After enter the subroutine service for software breakpoint, clear DBG_CR.SBRK
3. When service finish, use RTE instruction to return to the normal execution stream

Note that hardware can assure that the instruction following the instruction SBRK which causes exception will be really executed after the responsive handler routine finish.

7.6.4 Asynchronous Break Debug Exception

Asynchronous break is similar to INT request, but its privilege level is less than INT request. Once an asynchronous break is granted, hardware takes the same measure as fetch breakpoint, and then the following actions maybe done:

For hardware (necessary):

1. Load common debug handler entry from exception vector table (base + H'8)
2. Save returning location value to DPC
3. Save SR to DSR
4. Set value 0 to SR.DE to avoid reenter problem in debug handler
5. Set value 1 to SR.DS and SR.SM

For software (recommend, not necessary):

1. Read DBG_CR.BKEN to check that exact asynchronous break ^{*1} is granted
2. Cease asyn_brk_n^{*2}
3. Check whether DBG_CR.BKEN has been inactive
4. When service finish, use RTE instruction to return

Notes:

1. ^{*1} denotes that since asynchronous break maybe arises concurrently with synchronous fetch breakpoint, hence handler should investigate DBG_CR.IHIT. If both DBG_CR.IHIT and DBG_CR.BKEN are set, the asynchronous break should be responded first.
2. ^{*2} denotes that only in the host debugging environment, assert a write access to JTAG memory address H'FFFFFF20 to inform host that the asynchronous break is granted by CPU.

7.6.5 Asynchronous Boot Debug Exception

Asynchronous boot is similar to reset, but has highest priority. Once an asynchronous boot is granted, all instructions in the pipeline are cancelled, and then the following actions maybe done:

For hardware (necessary):

1. Load asynchronous handler entry from H'EC000000
2. Save current PC register value to DPC
3. Save current SR register value to DSR
4. Set value 0 to SR.DE
5. Set value 1 to SR.DS and SR.SM

For software (recommend, not necessary):

1. Invalidate D-cache, I-cache, D-TLB, I-TLB, GRF
2. etc.

7.7 Example For Application

7.7.1 Single step execution

```
...                               ! Following code sections run in supervisor mode
RCR R1, SR                       ! Read SR
ANDI R1, R1, 0x7D                ! Clear SR.DE to forbid break exception responding
WCR SR, R1                       ! Clear done
ORI R3, R0, -1                   ! R3 ← 0xFFFFFFFF
CST R3, 3, 1                     ! Mask DBG_IA0 fetch address comparison
CLD R2, 3, 0                     ! Read DBG_CR
ORI R2, R2, 0X0801               ! Enable DBG_CR.IDEN, DBG_CR.IMEN, DBG_CR.ASIDM
CST R2, 3, 0                     ! Enable done
RCR R1, ESR                      ! Read ESR
ORI R1, R1, 2                    ! Set value 1 to ESR.DE
WCR ESR, R1
RTE                             ! EPC contains the entry of the expected code
                                ! segment and a fetch breakpoint can be triggered
                                ! after each instruction in the target code segment
                                ! is executed
```

Target_section:

```
xxxx                             ! dbg_irq_n arise when this instruction flows to IF stage
YYYY                             ! dbg_irq_n arise when this instruction flows to IF stage
zzzz                             ! dbg_irq_n arise when this instruction flows to IF stage
...
```

Notes:

1. In the target code section, if the current ID stage instruction is a branch instruction (BCC or JA, J), dbg_irq_n should be granted until the branch instruction flows to EX stage.
2. The fetch breakpoint handler must clear DBG_CR[IHIT], otherwise the executing would be blocked at the first instruction of the target section, because the instruction triggers fetch breakpoint after RTE each time.

7.7.2 Combinatorial Break Condition Capture

```

_LOOP:
...
Sx16 R1, R2, R3      ! if R1 == 1, it triggers data access result break
BEQ R4, R5, _target ! if R4 == R5, branch to _target
xxxx
YYYY
J R7, <_LOOP>        ! continue loop
xyxy
_target:
zyxy                 ! fetch this instruction should trigger a fetch
breakpoint
...

```

Suppose that above code section is located in a larger loop entity, and programmer wants to capture the case that both the data access result breakpoint and fetch breakpoint are triggered simultaneously, how to do it? Following code give a solution. Please note that such combinatorial break condition detection can not be automatically performed by hardware, which need software (debug handler) assistance.

```

...
RCR R1, SR           ! Following code sections run in supervisor mode
ANDI R1, R1, 0X7D    ! Read SR
WCR SR, R1           ! Clear SR.DE to forbid break exception responding
LHI R3, 0X018000     ! Clear done
ORI R3, R3, 0X400    ! 0X0C000400 is the _target address
CST R3, 3, 1         ! Set DBG_IA0
CST R0, 3, 2         ! Set DBG_IA1, DBG_IA0 need compare
ORI R4, R0, 0X88     ! 0X88 is the expected ASID
CST R4, 3, 6         ! Set DBG_ASID
CST R0, 3, 4         ! Set DBG_DA1, DBG_DA0 need compare
ORI R3, R3, 0XC00    ! 0X0C000C00 is the target data access address
CST R3, 3, 3         ! Set DBG_DA0
ORI R4, R0, 1        ! 0X1 is the target data access result
CST R4, 3, 5         ! Set DBG_DD0
ORI R2, R0, 0X3C4B   ! Enable DBG_CR.IDEN, DBG_CR.DDEN, DBG_CR.ASIDM
                     ! DBG_CR.DMSK0
CST R2, 0, 0         ! Only the write half word access causing
                     ! result 0X1 can match
RCR R1, ESR          ! Read ESR
ORI R1, R1, 2        ! Set value 1 to ESR.DE
WCR ESR, R1
RTE                  ! EPC contains the entry of the expected code
segment
...

```

In the data access result breakpoint subroutine, if checking DBG_CR.IHIT[0] and DBG_CR.DHIT[0] represents that both of them are set, which means the combinatorial condition is match.

7.7.3 Data transfer between target and host

In host-monitoring debug, the data of JTAG memory are scanned into or out of JTAG interface. For different type, address[1:0], endian and access, data should be shifted into DATA registers or out from DATA register correctly. The following table lists the shift count in different conditions.

Type	Address[1:0]	big-endian		little-endian	
		read	write	read	write
byte	0	8	32	32	8
	1	16	24	24	16
	2	24	16	16	24
	3	32	8	8	32
half word	0	16	32	32	16
	2	32	16	16	32
word	0	32	32	32	32

Notes:

1. Read/write is viewed from target, that is to say, read is a LOAD instruction executed at target and write is a STORE instruction executed at target.
2. Here only lists the shift count of data. Another one shift is also needed to shift a "1" into DATA[32] to tell TAP that data access has been finished and bus is not frozen.

7.7.4 How To Access JTAG Memory Space

When the FSM is in Test-Logic-Reset because TRST_ is active or high level of TMS retains at least 5 TCK cycles, adopt the following algorithm to monitor load/store JTAG memory access issued by CPU. The example does not include burst access case.

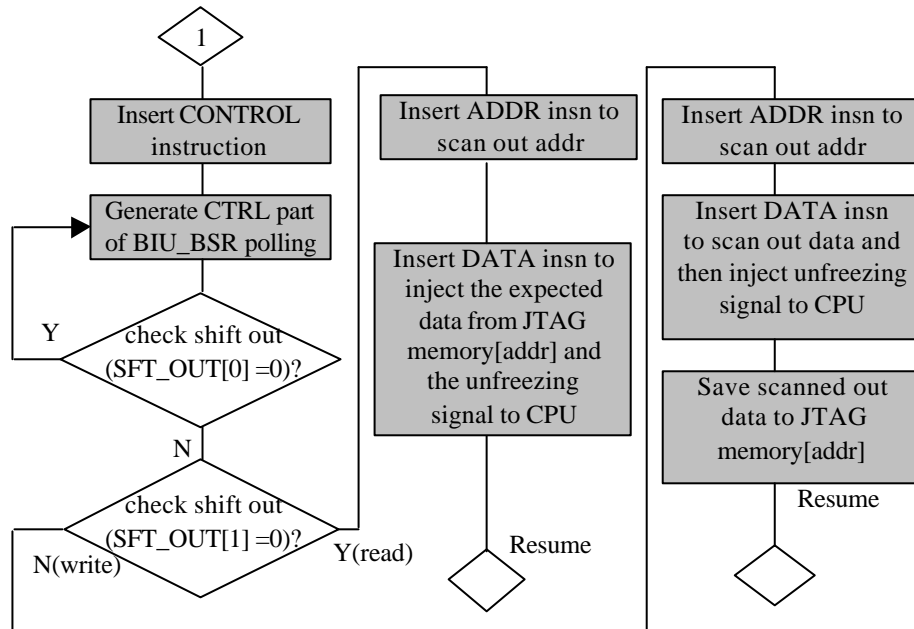
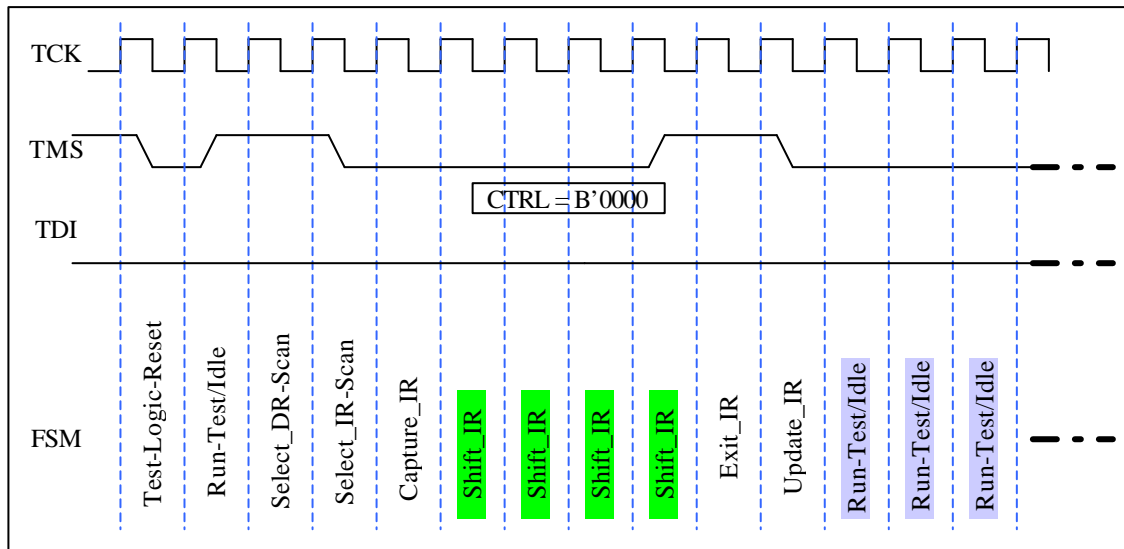


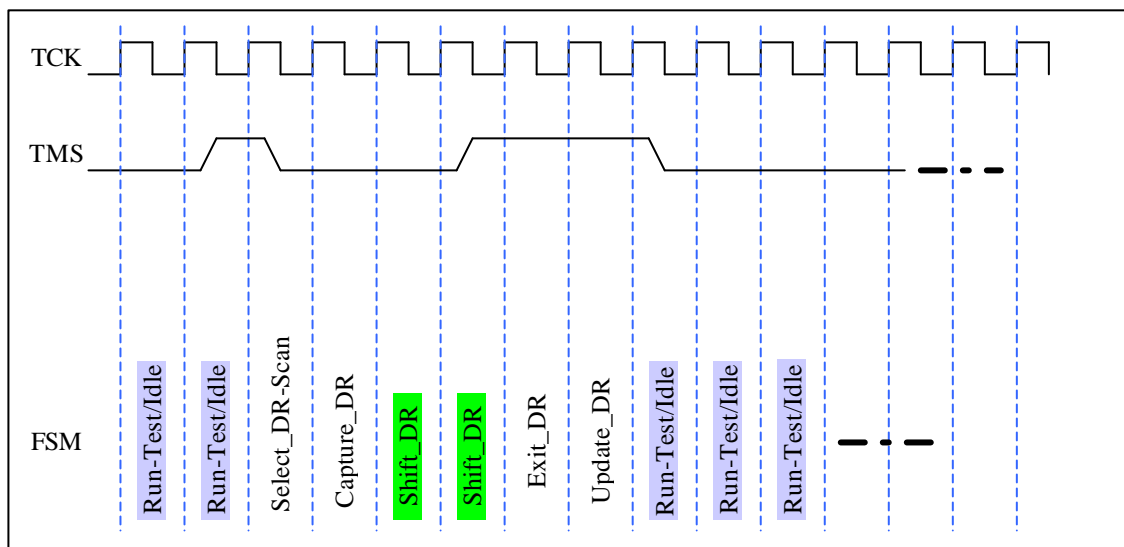
Figure 7-7 Debug and Extended JTAG

The following are correlative timing diagrams. **Although the processor supports two endians, the following figures only considers big-endian.**

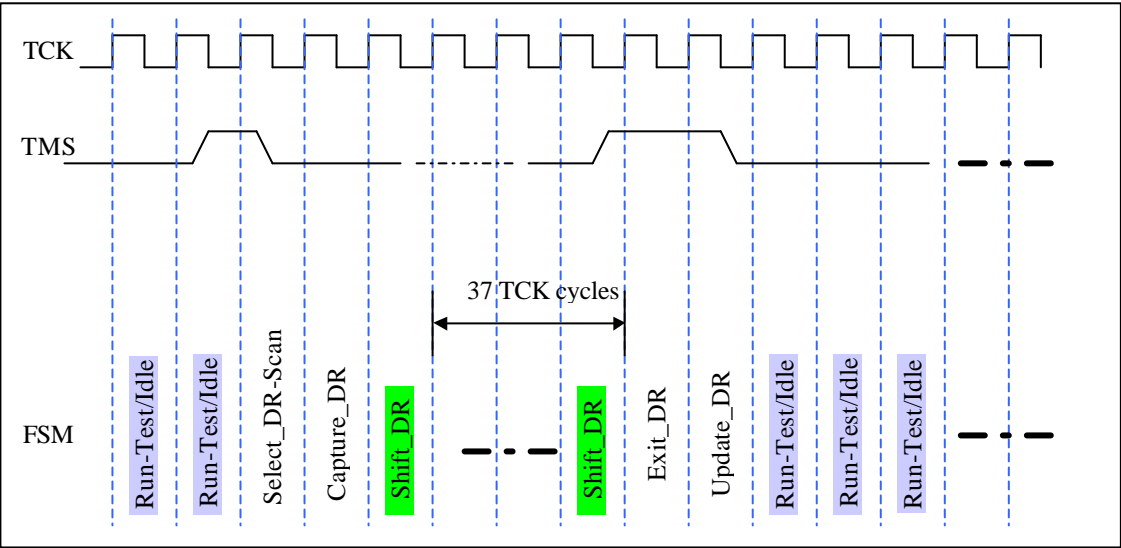
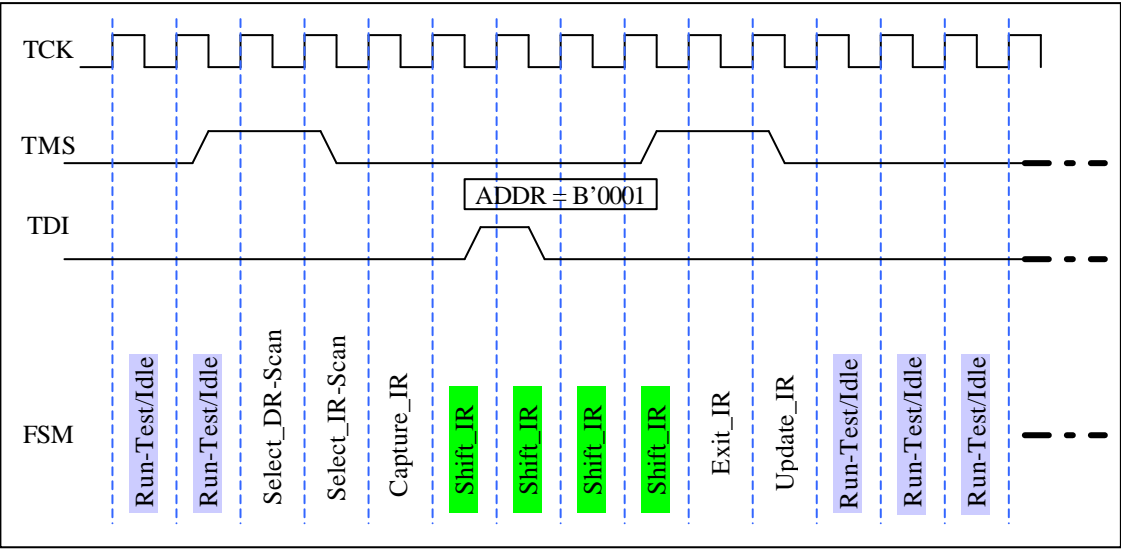
1. Insert CONTROL instruction to replace the default BYPASS instruction



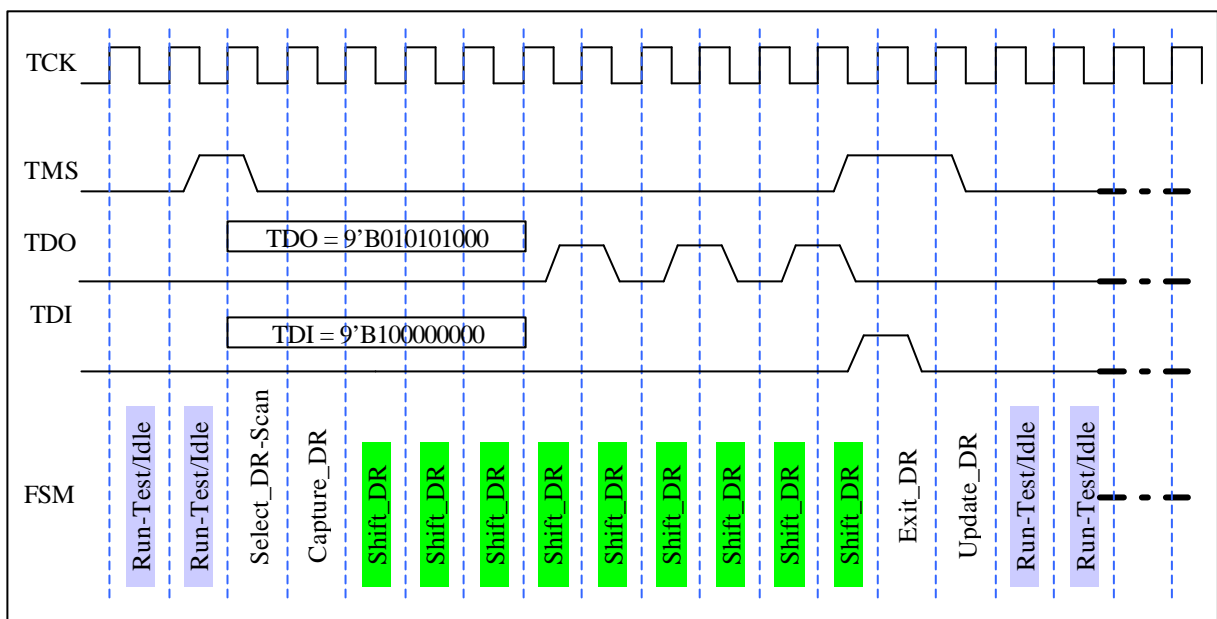
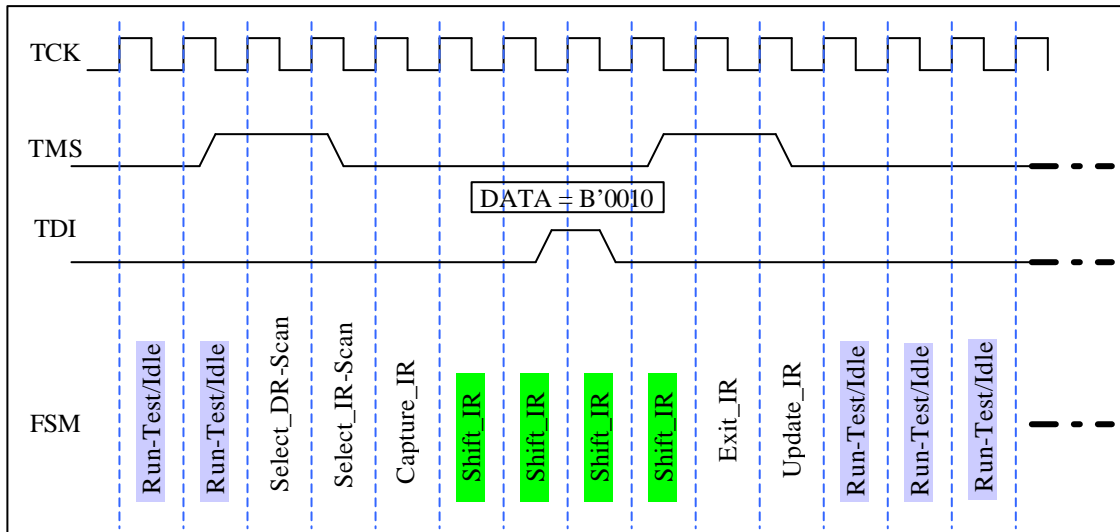
2. Use following TMS series to implement BIU_BSR.CTRL polling function



3. Use ADDR instruction to scan out ADDR part of BIU_BSR (for write JTAG memory access)

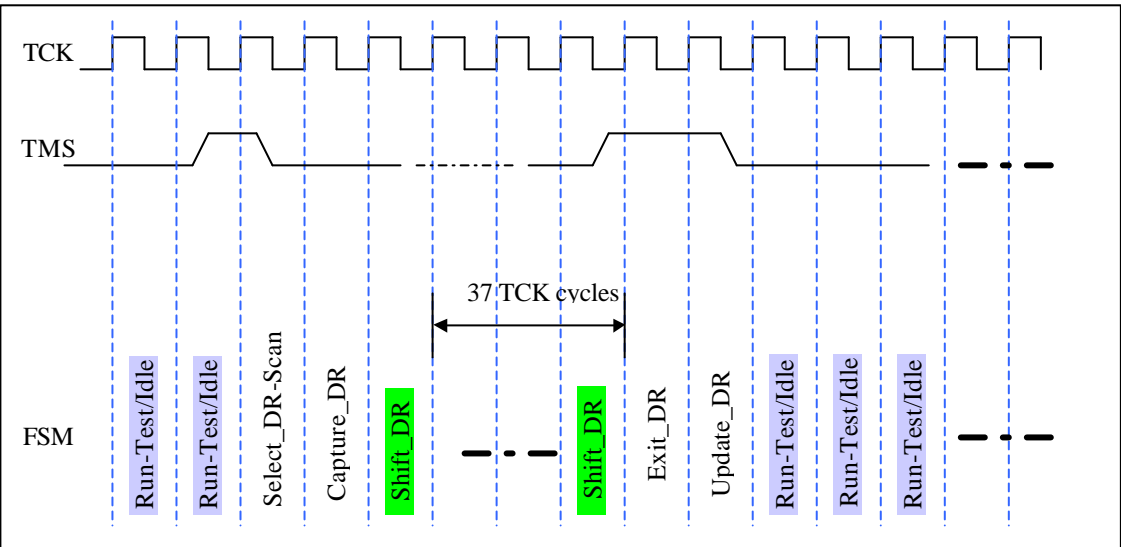
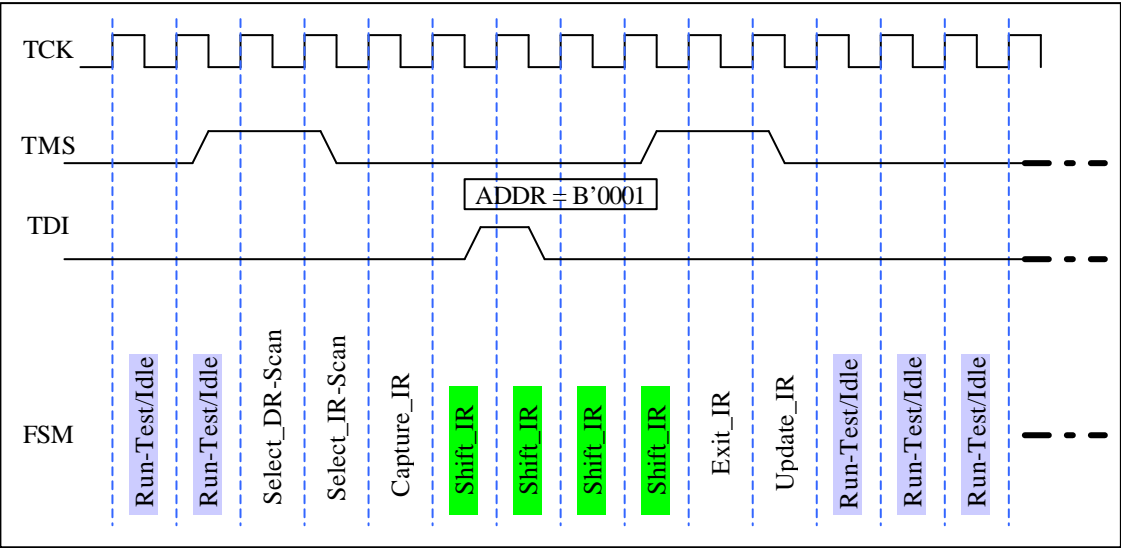


4. Use DATA instruction to scan out write data and inject unfreezing CPU signal (for write JTAG memory access)

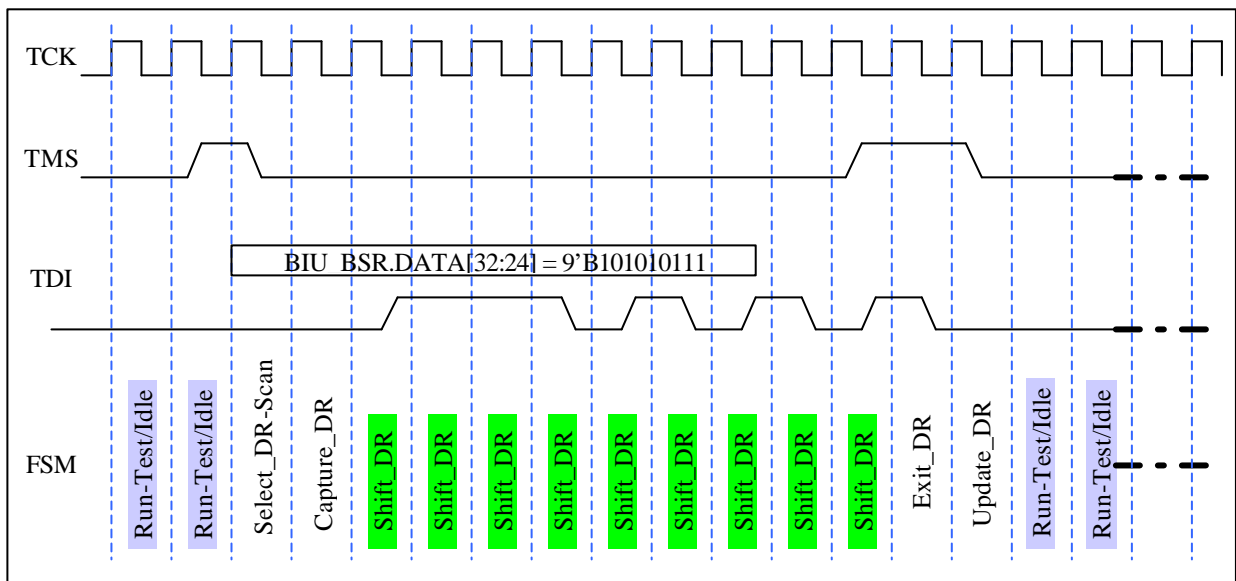
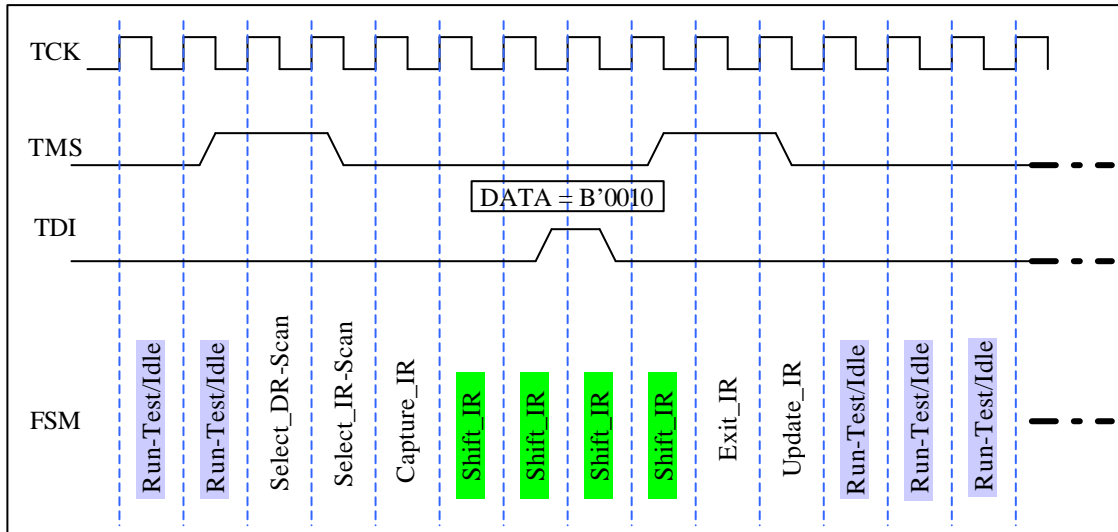


Note: In the case, it's a write byte access, and the value of `addr[1:0]` is 3, the write result is H'a8. For half word of word access, the circulating times of Shift_DR need make corresponding extension. And the alignment for big endianness must be cautious, for example, if CPU wants to write a byte value to H'EC000080, it needs 33 times of shifting, and the TDI must be 33'B1x. (x means don't care bits, which has 32 bits long and can be 0 or 1).

5. Use ADDR instruction to scan out ADDR of BIU_BSR (for read JTAG memory access)



- Use DATA instruction to inject expected data and unfreezing CPU signal (for read JTAG memory access)



Note: In the case, it's a read byte access, and the value of addr[1:0] is 0, the read result is H'57. For half word or word access, the circulating times of Shift_DR need make corresponding extension. And the alignment for big endianness must be cautious, for example, if CPU wants to read a byte value H'73 from H'EC000083, the final shift data must be 33'B1x01110011. (x means don't care bits, which has 24 bits long and can be any value).

7.7.5 How To Implement Burst Access from JTAG memory (burst read 8 words / burst write 4 words)

Use following algorithm, and the timing is similar (need some alteration) to above case.

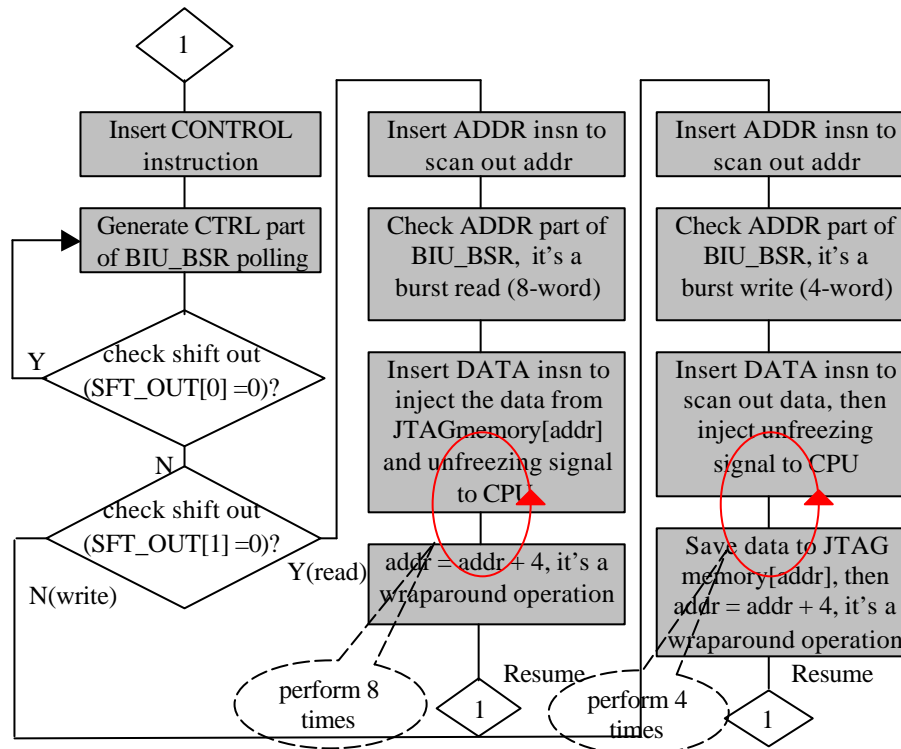


Figure 7-8 Debug and Extended JTAG

Note that burst access from JTAG memory uses wrap-round mode.

1. If burst access is write, which means writing 4 words. For example, if the burst access address is 0xEC000004, then the addresses of written data should be 0xEC000004, 0xEC000008, 0xEC00000C, 0xEC000000.
2. If burst access is read, which means reading 8 words. The addresses of read data are splitted into two groups: low group and high group. Wrap-round access only occurs in the group that has the start address and sequence access occurs in the group that has not the start address. For example, if the burst access address is 0xEC000004, then the addresses of read data should be 0xEC000004, 0xEC000008, 0xEC00000C, 0xEC000000, 0xEC000010, 0xEC000014, 0xEC000018, 0xEC00001C. if the burst access address is 0xEC000014, then the addresses of read data should be 0xEC000014, 0xEC000018, 0xEC00001C, 0xEC000010, 0xEC000000, 0xEC000004, 0xEC000008, 0xEC00000C.

7.7.6 How To Boot System From JTAG Memory

The case illustrates how to boot system from JTAG memory after power-on reset. It is very useful for developing system without on-board memory system at all. The steps are listed below:

1. Set TAP_SEL to 1 to select the internal TAP.
2. Press power key to reset processor and the internal TAP works.
3. Use TMS or TRST_ to reset the internal.

4. Inject ASYN_BOOT instruction from JTAG interface into the instruction register of the internal TAP
5. The boot handler entry located at H'EC000000 is load by CPU, hence system can boot from JTAG memory space successfully.

8 Performance Monitor

8.1 Overview

CPU performance is the guidance in evaluation of ISA, micro-architecture, cache and MMU. In addition, it can supply information for compiler writers, system developers and software programmers.

Arca2 provides two 32-bit performance counters that allow two unique events to be monitored. In addition, a 32-bit clock counter can be used with the performance counters. When any one of the three counters reaches its maximum value 0xFFFFFFFF, an overflow interrupt will occur. At the same time, the corresponding counter will wrap to zero and continue counting.

8.2 Register Configuration

PMON registers are listed in Table 8-1 and they can be accessed by CLD/CST instructions.

Table 8-1 PMON Registers

Name	Full Name	R/W	Initial value when power on	Access Size	#ID	#CR
PMC	Performance Monitor Control Register	R/W	H'00003FFC*	32	001	000
CTR	Clock Cycle Time Register	R/W	H'00000000	32	001	001
MOR0	Monitor Object Counter Register 0	R/W	H'00000000	32	001	010
MOR1	Monitor Object Counter Register1	R/W	H'00000000	32	001	011

Note that the value of PMC is reset to H'00003FFC, but it will be H'00003FF0 next cycle after reset. So its value should be considered to be H'00003FF0 after reset.

8.2.1 Performance Monitor Control Register (PMC)

#ID=001 # CR =000

Bit:	31	30	29	28	27	26	25	24
Read:								
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	23	22	21	20	19	18	17	16
Read:							SM	flag
Write:								
Reset:	0	0	0	0	0	0	0	0
Bit:	15	14	13	12	11	10	9	8
Read:	flag		MOR1_SLT					MOR0_SLT
Write:								
Reset:	0	0	1	1	1	1	1	1
Bit:	7	6	5	4	3	2	1	0
Read:	MOR0_SLT				P	C	M	E
Write:								
Reset:	1	1	1	1	1*	1*	0	0

Bit 31 ~ Bit 18: Reserved bits, ignored in write operation, always 0 in read operation.

- **E:** Clock Counter Enable
0: clock counter is disabled.
1: clock counter is enabled.
- **M:** Monitor Counter Enable
0: all monitor counters are disabled.

1: all monitor counters are enabled.

– **C:** Clock counter reset

0: no action.

1: reset clock counters to 0x0. **This bit will be auto-reset to 0 at next clock.**

This bit will reset to '1' when system reset so that CTR will auto reset to zero at the same time.

This bit also clears overflow flag (bit 16).

– **P:** Performance Counter Reset

0: no action.

1: reset all monitor object counters to 0x0. **This bit will be auto-reset to 0 at next clock.**

This bit will reset to '1' when system reset so that MOR0 and MOR1 will auto reset to zero at the same time.

This bit also clears overflow flag (bit 15 ~ 14).

– **MOR0_SLT (Bit 8-Bit 4):** identify the source of events for first monitor counter.

This is **MOR0** selector and its value is listed in Table 8-3. It is unpredictable when system reset.

– **MOR1_SLT (Bit 13-Bit 9):** identify the source of events for second monitor counter.

This is **MOR1** selector and its value is listed in Table 8-3. It is unpredictable when system reset.

– **flag (Bit 16-Bit 14):** overflow flag

Bit 16: clock counter overflow flag

Bit 15: object counter1 overflow flag

Bit 14: object counter0 overflow flag.

Read value: 0: no overflow.

1: overflow has occurred.

Write value: 0: clear this bit.

1: no change.

– **SM (Bit 17):** This bit is used to select the monitored event.

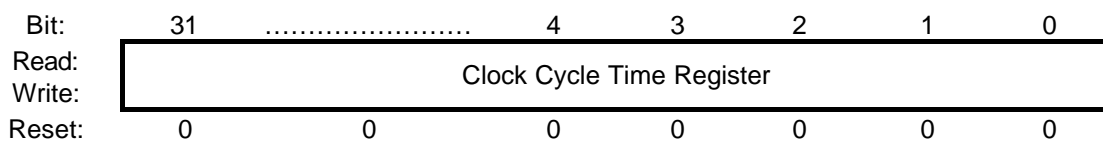
0: select all events in supervisor or user mode.

1: only select events which occur in supervisor mode.

8.2.2 Clock Cycle Time Register (CTR)

32-bit counter used to record clock cycles.

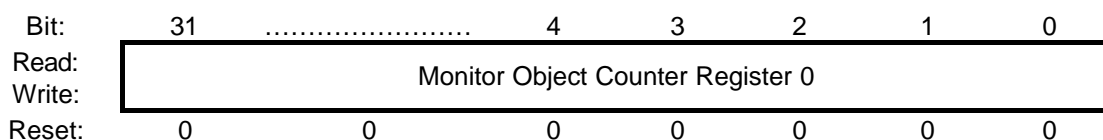
#ID=001 #CR=001



8.2.3 Monitor Object Counter Register 0 (MOR0)

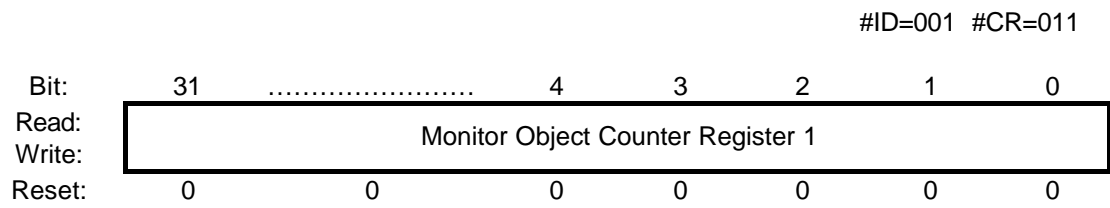
32-bit counter used to record the number of expected event such as TLB miss, Cache miss, jump and etc.

#ID=001 #CR=010



8.2.4 Monitor Object Counter Register 1 (MOR1)

32-bit counter used to record the number of expected event such as TLB miss, Cache miss, jump and etc.



8.3 Monitoring Event

The 5-bit MOR0_SLT and MOR1_SLT in PMC contain 32 events space respectively. It is divided to 4 parts as Table 8-2 lists.

Table 8-2 PMON Monitored Modules

MOR_SLT Highest 2-bits	Monitored Modules
00	IU
01	MMU and I-Cache
10	D-Cache
11	Reserved

Table 8-3 lists events that can be monitored by the MOR0/MOR1. Monitor Object Count Registers (MOR) can count any listed event. Software can select which event will be counted by setting the MOR0_SLT and MOR1_SLT fields of the PMC register.

Table 8-3 PMON Monitored Events

MOR_SLT		Event Name	Event Description
high	low		
00 (IU)	000	insn	Instruction that is executed. It also count those are canceled by exception.
	001	insn_cancel	Instruction that canceled by exception. It is not accurate if the instruction occupied more than one CPU pipeline stages
	010	stall	CPU pipeline stall, including canceled one
	011	sv_insn	Instruction that is executed in supervisor mode
	100	bcc_insn	BCC instruction, including canceled one
	101	bcc_taken	BCC instruction that branch is taken, including canceled one
	110	jump_insn	J, JA, RTE instruction, including canceled one
	111	Reserved	
01 (MMU I-Cache)	000	itlb_miss	ITLB miss count
	001	dtlb_miss	DTLB miss count. It also includes initial write exception.
	010	Reserved	
	011	Reserved	
	100	lcc_miss	Icache miss count, including cacheable fetch miss and uncacheable fetch miss.
	101	lcc_unc_fth	Icache fetch uncacheable area, which always induces icc miss.
	110	Reserved	
	111	Reserved	
10 (D-Cache)	000	Dcc_ldst	D-cache accepts load/store/swap command count, including fault command and the load when IU accept exception.
	001	Dcc_miss	D-cache does fill from external memory count.
	010	Dcc_unc	D-cache does single (read/write/read-then-write) access to external memory. It does not include fault access and bufferable write and missed write in write-through mode.
	011	Dcc_wbb	D-cache write back words count.
	100	Reserved	
	101	Reserved	
	110	Reserved	
	111	Reserved	
11 (Reserved)	000		
	001		

MOR_SLT		Event Name	Event Description
high	low		
	010		
	011		
	100		
	101		
	110		
	111		

By setting object event number in PMC and capturing different event counter, different performance can be monitored. Table 8-4 shows some typical combinations of monitored events.

Table 8-4 Typical Monitored Events

Items	MOR0_SLT	MOR1_SLT
CPI	0x0 (instruction count)	0x1 (instruction canceled)
Jump rate	0x5 (conditional jump that taken)	0x6 (unconditional jump)
BCC taken rate	0x5 (branch taken)	0x4 (branch instruction)
Stall rate	0x2 (stall)	0x0 (instruction count)
ITLB miss rate	0x8 (itlb miss)	0x0 (instruction count)
DTLB miss rate	0x9 (dtlb miss)	0x10 (load/store count)
I-Cache miss rate	0xC (I-cache miss)	0x0 (instruction count)
D-Cache miss rate	0x11 (D-cache miss)	0x10 (load/store count)
Data access rate	0x10 (dcc instruction)	0x0 (instruction count)

- CPI: cycles per instruction.

$$\text{CPI} = \text{CTR} / \text{MOR0}$$
- Jump rate: the rate of jumping target in total fetched instructions.

$$\text{jump_rate} = (\text{MOR0} + \text{MOR1}) / (\text{instruction count})$$
- BCC taken rate: the taken (jumping target) rate of BCC.

$$\text{bcc_taken_rate} = \text{MOR0} / \text{MOR1}$$
- Stall rate: how many stall happen when one instruction is executed

$$\text{stall_rate} = \text{MOR0} / \text{MOR1}$$
- ITLB miss rate: the rate of ITLB miss in total fetched instructions.

$$\text{itlb_miss_rate} = \text{MOR0} / \text{MOR1}$$
- DTLB miss rate: the rate of DTLB miss in total load/store instructions.

$$\text{dtlb_miss_rate} = \text{MOR0} / \text{MOR1}$$
- I-Cache miss rate: the rate of I-Cache miss in total fetched instructions.

$$\text{icache_miss_rate} = \text{MOR0} / \text{MOR1}$$
- D-Cache miss rate: the rate of D-Cache miss in total load/store instructions.

$$\text{dcache_miss_rate} = \text{MOR0} / \text{MOR1}$$
- Data access rate: the rate of D-Cache access in total fetched instructions.

$$\text{data_access_rate} = \text{MOR0} / \text{MOR1}$$

8.4 Monitoring flow

PMON monitors the object events when program is running. Two 32-bit MOR and one 32-bit CTR accumulate events independently before wrapping around. An overflow interrupt will occur when the counters wrap. Extended event logging may be accomplished by periodically reading the contents of the MOR0/MOR1/CTR before each overflow.

The steps using PMON can be concluded as follows.

1. Reset all PMON counters by setting P and C, clearing MOR_SLT and overflow flag bits in PMC.
2. Set MOR_SLT0 and MOR_SLT1 with object event number listed in Table 8-3 and SM bit (If only want to monitor events in supervisor mode, then set it to 1) .
3. Turn on clock counter and monitor counters by setting E bit and M bits in PMC.
4. If overflow interrupt occurs, record overflow in the corresponding variables.
5. Check the result of CTR, MOR0, MOR1 and overflow variables after program is over.

List of Figures

Figure 1-1 Arca2 CPU core Block Diagram	2
Figure 2-1 General pipeline	4
Figure 2-2 Exception hazard.....	7
Figure 3-1 Exceptions and Exception Resources	10
Table 4-1 Module Identification Number	18
Table 4-2 Control Registers in Module MMU.....	18
Table 4-3 CR in Module Debug.....	18
Table 4-4 Control registers in Module PMON	19
Table 4-5 Module CMD Definitions	19
Figure 5-1 Direct Virtual Physical Address translation (MCR.ATE=0).....	26
Figure 5-2 Paging Virtual Physical Address translation (MCR.ATE=1)	28
Figure 5-3 Configuration of TLB	30
Figure 5-4 Flowchart of Data Access Using DTLB	32
Figure 5-5 Flowchart of Instruction Fetch Using ITLB	33
Figure 6-1 Data Cache Structure.....	42
Figure 6-2 Instruction Cache Structure	45
Figure 7-1 Debug and Extended JTAG.....	58
Figure 7-2 Internal Tap.....	59
Figure 7-3 BIU_BSR Register	61
Figure 7-4 Endian Adjustment.....	62
Figure 7-5 The Stutcture of Debug Module	64
Figure 7-6 Exception Priority.....	74
Figure 7-7 Debug and Extended JTAG.....	81
Figure 7-8 Debug and Extended JTAG.....	87

List of Tables

Table 1-1 Arca2 CPU Core Features	3
Table 2-1 Instruction and special IU states cycles	9
Table 2-2 Stall conditions and cycles.....	9
Table 3-1 Arca Exception Priorities.....	12
Table 3-2 Arca Exception Vector Table.....	13
Table 3-3 Exception Cause.....	14
Table 5-1 MMU Registers	22
Table 5-2 MMU configuration instruction	34
Table 6-1 Cache Feature	39
Table 6-2 Cacheable and Bufferable attribute of one page	41
Table 6-3 D-cache and Write Buffer Policy.....	41
Table 6-4 Cache Operations	51
Table 7-1 Debug Features	57
Table 7-2 Extended JTAG Features	57
Table 7-3 Extended JTAG Instructions	60
Table 7-4 BIU_BSR Register.....	61
Table 7-5 JTAG Memory Space.....	62
Table 7-6 Debug Internal Signals	64
Table 7-7 Debug Registers	65
Table 7-8 Fetch Breakpoint True Table.....	71
Table 7-9 Data Access Breakpoint True Table	72
Table 7-10 Instructions That Are Not Monitored	73
Table 8-1 PMON Registers	90
Table 8-2 PMON Monitored Modules.....	93
Table 8-3 PMON Monitored Events	93
Table 8-4 Typical Monitored Events	94